REALIZZAZIONE DI ALCUNE STRUTTURE DATI DI BASE

Tabelle Hash

Heap Binomiali

B-Alberi Binari

MANUALE D'USO CON ESEMPI DI UTILIZZO

Maria Luisa Guerriero

9 dicembre 2001

Indice

Parte I

TABELLE HASH

Capitolo 1

Introduzione

Tra gli argomenti più trattati nella bibliografia del software ci sono sicuramente le hash table. Le hash table suscitano infatti ancora adesso molto interesse, nonostante il fatto che ogni libreria di funzioni o di classi seria ne contenga già un'implementazione, ed il motivo è questo: è facile costruire una hash table che si comporti meglio di un contenitore sequenziale come una lista o un array, tuttavia esaminandone le diverse opzioni di realizzazione si trovano diverse possibilità, nessuna delle quali è ottima in tutte le circostanze. Inoltre, non esiste un test di riferimento "ideale": combinando le possibilità di configurazione e di implementazione con i tipi di test che si possono fare, si ottengono risultati talmente disparati che è impossibile fornire una hash table adatta ad ogni problema.

Ma a cosa possono servire le hash table? Il loro più semplice utilizzo è quello di realizzare un **dizionario**, che è una struttura in cui si possono inserire, cancellare e ricercare degli elementi.

Vedremo come si può realizzare un dizionario in modo che l'operazione di inserimento costi $O(1)$ e quelle di ricerca e cancellazione costino $O(N/M)$, dove N è il numero di elementi memorizzati ed M è il numero di caselle della tabella.

Va però detto che ci sono molte alternative alla soluzione scelta, sia realizzando diversamente la tabella hash, sia utilizzando altre strutture dati. Nel seguito accennerò ad alcune di queste possibilità.

Inizierò col presentare un esempio di hash table e le operazioni che si possono effettuare su di essa. I capitoli successivi trattano i due temi principali da affrontare nell'implementazione di una hash table: la scelta di una funzione hash e la tecnica da usare per gestire le collisioni. In seguito esaminerò le caratteristiche di una hash table "ideale", ed infine presenterò la mia implementazione in C++.

1.1 Cos'è una Tabella Hash: un esempio

Una tabella hash è semplicemente un array che viene indicizzato per mezzo di una **funzione hash**, che converte una chiave in un valore intero usato come indice. Per esempio, nella Figura 1.1, `HashTable` è un array di 8 elementi. Ogni elemento è un puntatore ad una lista concatenata di dati numerici. La funzione hash, in questo esempio, non fa altro che dividere per 8 il valore da memorizzare, e usa il resto della divisione intera per indicizzare la tabella. Il risultato è un numero fra 0 e 7. Siccome l'intervallo degli indici per `HashTable` varia fra 0 e 7, siamo certi che l'indice ottenuto è valido.

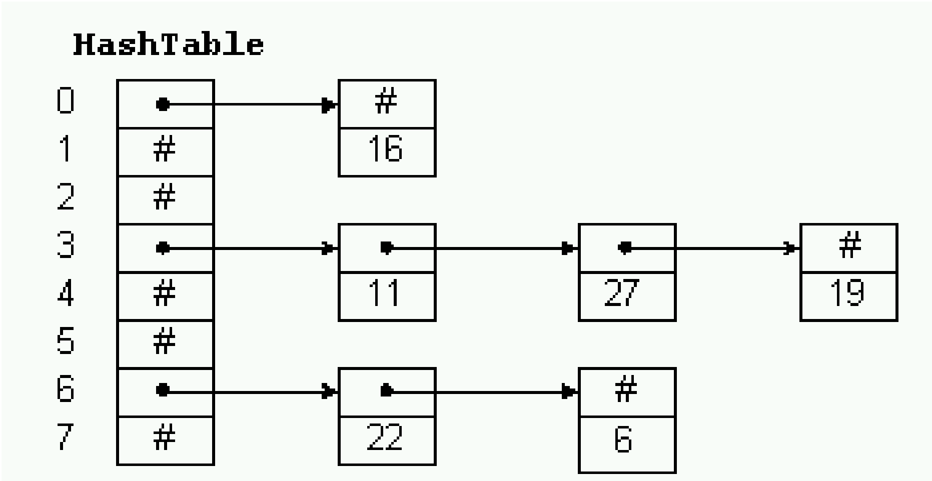

Figura 1.1: Una Tabella Hash

Per inserire un nuovo elemento nella tabella, applichiamo la funzione di hash al suo valore per sapere in quale lista esso vada inserito, e quindi lo inseriamo in testa alla lista. Per esempio, per inserire il valore 11, dividiamo 11 per 8 ottenendo un resto di 3. In questo modo 11 va nella lista che inizia da `HashTable[3]`.

Per trovare un numero, applichiamo la funzione di hash al suo valore e percorriamo la lista corrispondente per vedere se si trova nella tabella.

Per cancellare un numero, lo troviamo, ed eliminiamo il nodo in cui si trova dalla lista concatenata. In questo esempio la funzione hash usata adotta il metodo della divisione, e gli elementi memorizzati nella hash table sono contenuti in liste (o bucket); questa è solo una delle possibili soluzioni che si possono adottare per realizzare una hash table: vedremo altri esempi nei capitoli successivi.

Vediamo ora un possibile utilizzo delle hash table: contare quante volte venga ripetuta una parola all'interno di un file di testo.

Per maggiori informazioni sulle operazioni che si possono effettuare sulla hash table fornita, si rimanda al capitolo successivo o al 6, che contiene l'implementazione.

1.2 Un esempio pratico: CountWords.cpp

7 ⟨*CountWords.cpp* 7⟩≡

```
/*******************************************************
 *           TABELLE HASH CON LISTE DI TRABOCCO        *
 *  File di esempio d'uso del dizionario realizzato da  *
 *     ----------- Maria Luisa Guerriero ----------     *
 *******************************************************/
/* -----------------------------------------------------
```

```
* Il file CountWords.cpp prende in input uno o piu' file di testo
  (passati come argomento al programma), o lo standard input, e
  mostra le parole lette ed il numero di volte in cui sono state
  incontrate.
* Le parole sono definite semplicemente come serie di caratteri,
  separati da spazi.
------------------------------------------------------- */

#include <fstream>
#include <string>

#include "Table.h"

//----------------------------------------------------------

void countfile(istream &inf)
{
 Table<string> T;
 string word;

 T.InitTable();

 while (inf >> word)
   T.Insert(word);

 T.ShowTable();
}
//----------------------------------------------------------

int main (int argc, char **argv)
{
 int exitstatus = 0;

 if (argc > 1)
 {
   for (int i = 1; i < argc; ++i)
   {
     ifstream inf(argv[i]);

     if (!inf)
     {
       cout <<"Impossibile aprire il file "<< argv[i] << endl;
       exitstatus++;
       continue;
     }
     cout << "\nfile " << argv[i] << "\n\n";
     countfile(inf);
   }
 }
 else countfile (cin);
```

```
 return (exitstatus);
}
//-------------------------------------------------------
```

1.3 Operazioni sulla hash table

Le funzioni supportate dalla hash table sono:

- **Table** - Costruttore: crea una hash table:
  O(1) → Crea un puntatore ad un array di liste
- **InitTable** - Inizializza la hash table:
  O(M) → Crea M liste
- **Insert** - Inserisce un elemento nella hash table:
  O(L) → Inserisce in testa alla lista dopo aver controllato che non sia già presente
- **ShowTable** - Mostra il contenuto della hash table:
  O(M·L) → Scandisce tutte le M liste
- **Delete** - Cancella un elemento dalla hash table:
  O(L) → Scandisce la lista associata al valore hash dell'elemento
- **Find** - Cerca un elemento nella hash table:
  O(L) → Scandisce la lista associata al valore hash dell'elemento

dove M è il numero di buckets ed L è la lunghezza della lista di lunghezza massima. Il valore di L dipende dalla distribuzione degli elementi.

Capitolo 2

Un fattore importante: la dimensione

Come dimostreremo in seguito, la ricerca di un elemento nella tabella ha complessità nel caso pessimo pari a $O(N)$, dove N è il numero di elementi memorizzati al suo interno: non sembrerebbe niente di eccezionale! Vedremo però anche che le prestazioni sono ottime nel caso medio, perché la complessità diventa $O(1)$, quindi è costante (contro un tempo proporzionale a $\log N$ nel caso degli alberi binari). Il problema delle hash table è che tararle a dovere è un'operazione difficile che richiede informazioni che spesso non sono disponibili, come il numero massimo di elementi che andranno nella hash table e la distribuzione delle loro chiavi hash. Consideriamo, per esempio, una tabella usata come contenitore delle parole trovate in un frammento di testo: anche se la funzione di hash fosse "perfetta", nel senso che ogni sequenza possibile di lettere avesse uguale probabilità di finire in uno qualunque delle celle della tabella, resta il fatto che le parole in qualunque lingua non hanno una distribuzione uniforme! Quindi esiste il rischio che una funzione hash teoricamente perfetta finisca per concentrare le parole incontrate in poche posizioni. Non c'è quindi da stupirsi se la ricerca della funzione hash ideale sia praticamente impossibile nè del fatto che test diversi sulle stesse funzioni di hash riportino risultati del tutto contrastanti.

Per prima cosa bisogna decidere la **dimensione** della tabella. La chiave hash finale è il modulo (resto) della divisione tra il valore di hash e la dimensione della hash table. La maggiore dispersione del modulo avviene quando il valore di hash e la dimensione della hash table sono reciprocamente primi, il che può essere assicurato solo se la dimensione della hash table è un numero primo. In [3] Bistock afferma che portando la dimensione di una hash table da 30.240 bucket (che è divisibile per tutti gli interi da 2 a 9) a 30.241 bucket (che è un numero primo) e immettendo 42,829 parole uniche, la lunghezza massima dei bucket passa da 10 a 6.

Posto quindi che la dimensione della tabella hash debba essere un numero primo, nello scegliere il valore di questo numero, bisogna cercare un compromesso fra memoria e velocità di esecuzione delle operazioni. Ovviamente più la tabella è grande, più è veloce la ricerca di elementi al suo interno.

Come mostra la tabella seguente, una tabella di piccole dimensioni aumenta apprezzabilmente il tempo medio per la ricerca di un valore. Una tabella hash può pensarsi per ora come una collezione di liste concatenate. Con l'aumentare della dimensione della tabella, aumenta il numero di liste, e diminuisce il numero medio di nodi per ogni lista. Una tabella di dimensione unitaria corrisponde, di fatto, ad un'unica lista concatenata di lunghezza N (dove N è il numero di elementi inseriti). Assumendo di usare una funzione di hash perfetta, una tabella di dimensione 2 avrà 2 liste di lunghezza $N/2$. Se la dimensione della tabella è 100, si avranno 100 liste di lunghezza $N/100$, riducendo considerevolmente la dimensione delle liste in cui effettuare le ricerche. La scelta della dimensione della tabella è perciò di notevole influenza.

Dimensione	Tempo	Dimensione	Tempo
1	869	128	9
2	432	256	6
4	214	512	4
8	106	1024	4
16	54	2048	3
32	28	4096	3
64	15	8192	3

Tabella 2.1: Confronto tra HashTableSize e tempo medio di ricerca (μs), 4096 elementi

Di solito però non è dato di sapere in anticipo quanti elementi faranno parte di una hash table, per cui si deve prevedere che essa incorrerà in una o più espansioni nel corso della sua vita. Il modo più semplice di espandere una hash table consiste nel riallocarla completamente, ricalcolare di conseguenza la funzione di hash con la nuova dimensione per tutti i dati dalla hash table e trasferirli da quella vecchia a quella nuova. Questo procedimento, detto resize intermittente o **ristrutturazione**, ha costi di tempo proporzionali alla dimensione della hash table, proibitivi per hash table grandi.

Esistono anche dei tipi di hash table che si espandono gradualmente (vedere [5] e [9]), evitando in questo modo il ricorso alla riallocazione completa; questo però comporta un aumento uniforme del costo di inserimento, per cui lo spreco di tempo in fin dei conti viene solo ripartito lungo l'intera vita della hash table anziché essere concentrato in pochi momenti.

Infine un'ultima considerazione: così come quando il fattore di carico supera una certa soglia si aumentano le dimensioni della hash table, quando il fattore di carico scende al di sotto di una soglia inferiore si dovrebbe, per coerenza, diminuirne le dimensioni. In realtà pochissime implementazioni di hash table eseguono questa operazione, sacrificando la memoria all'efficienza. In seguito a queste considerazioni, si capisce perché l'uso di una hash table in un programma sia preferibile agli alberi binari solo quando l'ottimizzazione delle prestazioni giustifica lo sforzo di tararla a dovere.

Capitolo 3

La funzione Hash

3.1 Cos'è l'hashing e perché si usa

Se ci si potesse permettere di avere tabelle di enormi dimensioni, ogni chiave potrebbe essere trasformata univocamente in un numero intero semplicemente convertendo ogni simbolo della chiave nella sua rappresentazione binaria e interpretando la stringa di bit come un indice della tabella: il record potrebbe essere così memorizzato al suo indirizzo "corretto". La ricerca richiederebbe un solo passo: trasformare la chiave e leggere il record dalla sua unica cella nella tabella. Questo ideale, però, non è realizzabile: per trattare, ad esempio, tutte le chiavi di 6 caratteri, con ogni carattere codificato con 8 bit, servirebbe una tabella con $2^{48} \simeq 2^{50} \simeq 10^{15}$ di celle (poiché $2^{10} \simeq 10^3$)!! E, anche se vi fossero memorizzati $N = 10^6$ elementi, verrebbero usate solo $10^6/10^{15} = 10^{-9}$, cioè 1 miliardesimo, delle celle!! Molto inefficiente per quanto riguarda l'uso della memoria!

Come compromesso pratico, si usano tabelle più piccole di quella descritta sopra, e si progetta una funzione di trasformazione delle chiavi, detta **funzione di hash**, che permetta (potenzialmente) a più chiavi di essere associate nello stesso indirizzo della tabella. L'obiettivo è quello di fare in modo che l'associazione delle chiavi agli indici della tabella sia il più uniforme possibile. Cioè, per quanto molte chiavi possano finire nella stessa posizione, ogni cella riceva, in media, all'incirca tante chiavi quante le altre. In questo modo avendo una tabella con M celle in cui sono memorizzate N chiavi, circa N/M record saranno associati idealmente ad ogni casella. Ovviamente, nell'inserire una particolare sequenza di chiavi nella tabella, alcune caselle saranno vuote, altre conterranno pochi record, ed altre ancora ne conterranno molti. Quello si ottiene risulta quindi essere un eccellente tempo medio di ricerca.

I metodi hash basano il loro funzionamento su un calcolo matematico eseguito sulla chiave di ricerca. Tale calcolo ricava la locazione dei dati utilizzando la chiave come dato in input. La funzione hash deve essere semplice, calcolabile velocemente e deve ripartire uniformemente i valori calcolati. Due criteri primari sono costituiti dagli inserimenti efficienti di nuovi elementi e da ricerche veloci degli elementi esistenti.

Per fare un esempio, supponiamo che un libro venga archiviato per mezzo di un numero ISBN, come 0-19-502402-8. Se il numero ISBN viene trattato come un intero ed una semplice funzione hash viene definita come f(ISBN)=ISBN mod 500, la localizzazione di ogni libro all'interno di una tabella dei dati viene determinata immettendo il proprio numero ISBN all'interno della funzione. Poiché la funzione hash indicizza la tabella dei dati, l'intervallo dei valori in uscita deve corrispondere alle dimensioni della tabella. La funzione hash mostrata sopra possiede per sua natura un difetto, poiché nulla evita che la funzione selezioni la stessa locazione per due libri diversi; indici duplicati si verificano qualora due numeri ISBN differiscano per un multiplo di 500.

Valori hash coincidenti associati a chiavi differenti vengono chiamati **collisioni**. Si possono gestire le collisioni nei metodi hash in uno dei seguenti modi:

- prevenendo il verificarsi di collisioni, utilizzando una funzione hash perfetta
- utilizzando elaborazioni addizionali per gestire le collisioni, qualora avvengano

Funzioni hash perfette, che non producono collisioni, possono essere ricavate unicamente quando il numero delle chiavi è molto piccolo e le chiavi stesse sono fisse, come in una lista delle parole-chiave di un compilatore. Poiché le collisioni non possono essere evitate nella maggior parte delle applicazioni, gli inventori di funzioni hash si concentrano sulla riduzione della frequenza delle collisioni, producendo una distribuzione uniforme dei valori generati.

Se la funzione di hash è uniforme, cioè ripartisce uniformemente i valori fra gli indici della tabella di hash, allora l'operazione di hashing suddivide efficacemente i valori fra le liste in cui effettuare le ricerche. Il caso peggiore si verifica quando tutti i valori vengono fatti corrispondere allo stesso indice dalla funzione di hash e ci si trova ad avere un'unica lista concatenata che si deve percorrere sequenzialmente per effettuare le ricerche. Di conseguenza, è importante scegliere una buona funzione di hash.

La funzione hash che ho implementato è molto semplice e veloce da calcolare, ma non è di certo la migliore per quanto riguarda la distribuzione delle chiavi.

Per una rassegna delle varie tecniche di hashing vedere l'Appendice A.

Capitolo 4

Gestione delle collisioni

L'obiettivo principale da affrontare, dato che le collisioni sono inevitabili, è come gestire l'indirizzamento secondario dovuto al fatto che un solo record può risiedere in una casella.

Il fatto che le collisioni sono inevitabili anche con una funzione hash perfetta può essere chiarito con un esempio. Supponiamo di avere una tabella di dimensione M; qual è la probabilità di associare, ad esempio, 5 chiavi ad indirizzi distinti, cioè senza alcuna collisione? Se la trasformazione delle chiavi in interi è ideale ogni chiave ha la stessa probabilità di finire in ognuna delle caselle. La prima chiave va in una qualche casella; la seconda la "mancherà" con probabilità $1 - 1/M$; la terza le mancherà entrambe con probabilità $1 - 2/M$, e così via. La probabilità che si verifichino tutti questi eventi insieme è:

$$(1 - 1/M)(1 - 2/M)(1 - /3M)(1 - 4/M)$$

Per $M = 20$, ad esempio, la probabilità di inserire i 5 elementi senza collisione è $(0.95) \cdot (0.90) \cdot (0.85) \cdot (0.80) = 0.58$. Questo vuol dire che circa il 40% delle volte si avrà una collisione nell'inserire solamente $1/4$ degli elementi che può contenere la tabella. È quindi fondamentale gestire le collisioni, più che evitarle.

Ci sono due tipi principali di hash table, a seconda del metodo utilizzato per risolvere le collisioni e saranno oggetto delle 2 sezioni seguenti. Ci concentreremo in particolare sul **metodo a scansione esterna delle liste di trabocco**, che è quello che ho adottato nell'implementazione.

4.1 Metodi a scansione interna

Le Hash Table in cui viene usato uno di questi metodi si dicono **"chiuse"**. In queste caso ogni entry della tabella è un record singolo e in caso di collisione si applica una nuova funzione di hash. Tutti gli elementi sono memorizzati all'interno della tabella hash stessa: il **fattore di carico** $a = N/M$ perciò non supera 1; quando si tenta di inserire l'$M + 1$-mo oggetto, si raggiunge una condizione di overflow e bisogna adottare un comportamento speciale.

I metodi interni si distinguono essenzialmente per il modo in cui è fatta la scansione secondaria, cioè il modo in cui si trova la posizione ai record la cui chiave viene associata a una cella già occupata.

Lo schema più semplice di scansione secondaria è detto linear probe, o **scansione lineare a passo unitario**. Altri metodi sono: **scansione quadratica** e **hashing doppio**. Per qualche informazione in più su questi metodi vedere l'Appendice B.

4.2 Metodi a scansione esterna

Le Hash table che usano il **concatenamento** (o **chaining**) si dicono **"aperte"**. In questo caso ad ogni entry nella hash table (e quindi ad ognuno dei valori generabili dalla funzione hash) viene associato un bucket (una struttura dati in grado di contenere più di un elemento chiave ed un riferimento al blocco dati) che contiene tutti i record corrispondenti alla stessa chiave hash.

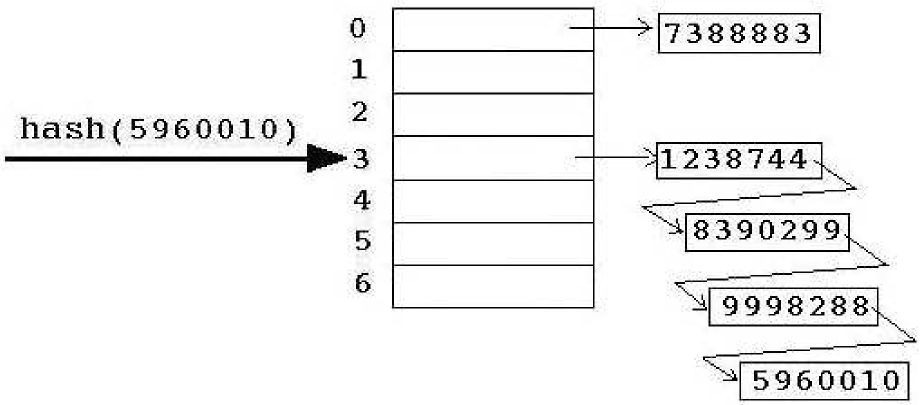

Figura 4.1: Tabella Hash con liste di trabocco

Con questo metodo due chiavi che collidono sulla stessa locazione vengono immagazzinate nello stesso bucket (Fig. 4.1), andando a formare una catena. Gli elementi, quindi, non sono memorizzati all'interno della tabella hash, ma nei bucket ad essa associati. Per trovare un elemento con una certa chiave, la funzione hash per prima cosa identifica il bucket corretto, poi confronta individualmente gli elementi contenuti nel bucket con la chiave di ricerca. Il numero di chiavi immagazzinato in ogni bucket è generalmente così piccolo da permettere di poter adeguatamente implementare un bucket tramite una lista concatenata (detta lista di trabocco). Se la funzione hash primaria è abbastanza uniforme, infatti, ci si aspetta che le liste delle locazioni attive siano lunghe circa N/M. Una ricerca sequenziale sulla lista permette quindi di identificare la chiave corretta.

Per trovare l'i-mo record di una lista di k record bisogna fare i confronti. Se assumiamo che la chiave cercata si possa trovare con uguale probabilità in ognuno dei k record della lista, il costo medio della ricerca di un elemento in una catena di k elementi è $(k+1)/2$ (ricerca sequenziale in una lista lunga k).

Supponendo di avere una funzione hash ideale, ci si aspetta che k sia circa $1+(N-1)/M$. Si sa, infatti, che la chiave da ricercare è da qualche parte nella lista e che le altre $N-1$ chiavi saranno idealmente distribuite uniformemente nelle M caselle della tabella hash. Perciò il costo medio di una ricerca con successo è

$$\frac{1+1+\frac{N-1}{M}}{2} \simeq 1+\frac{a}{2}$$

dove $a = N/M$ è il fattore di carico della hash table.

Questo è un risultato notevole considerando il fatto che anche per gli alberi binari bilanciati di ricerca il costo medio di ricerca è $O(\log N)$. Occorre però ricordare che il costo nel caso pessimo per questi è sempre $O(\log N)$, mentre il caso pessimo per le hash table con scansione esterna è $O(N)$: tutti gli elementi potrebbero finire nella stessa lista della tabella. L'idea dell'hashing sfrutta il fatto che il caso pessimo è estremamente improbabile e non contribuisce molto al costo medio.

Il costo di una ricerca con insuccesso è $a = N/M$ in media, dato che bisogna confrontare la chiave ricercata con tutte quelle presenti nella lista associata alla casella, scoprendo infine che nessuna coincide, e la lunghezza media della lista per una "buona" funzione hash è $a = N/M$.

In conclusione, la hash table permette di memorizzare un nuovo elemento in tempo costante $O(1)$: basta inserirlo in testa alla lista associata all'indice desiderato; inoltre si può, in media, trovare un elemento (o verificare che non appartiene alla tabella) in $O(N/M)$; il costo nel caso pessimo è comunque $O(N)$; la cancellazione è associata alla ricerca ed ha la stessa complessità.

Una limitazione delle hash table è che non esiste un modo semplice di costruire una lista ordinata degli elementi. Sarebbe possibile ridurre ulteriormente il costo di ricerca e cancellazione costruendo un albero binario di ricerca per ogni cella attiva della tabella, invece di una lista collegata; tuttavia l'overhead aggiuntivo degli alberi di ricerca potrebbe compensare ogni riduzione nel costo della ricerca.

Usando il metodo delle liste di trabocco il fattore di carico a può superare 1: si possono memorizzare più record di quante sono le celle della tabella hash. C'è però bisogno di spazio aggiuntivo in ogni record per i campi dei link.

Si noti che nei metodi a scansione esterna la catena di chiavi con lo stesso valore di hash è memorizzata esplicitamente, mentre in tutti gli schemi di scansione secondaria la catena è implicitamente presente nella sequenza iniziata dalla prima collisione, ed è collegata da computazioni successive.

Infine si può dimostrare che la lunghezza media di un bucket dipende solo dal fattore di carico della hash table, mentre la lunghezza massima è legata anche alle dimensioni assolute della tabella. Infatti più una hash table è grande e più è alto, a parità di probabilità, il numero di bucket che arriva ad una data lunghezza. Ad esempio una hash table di 100 bucket caricata con 200 elementi ha bucket di lunghezza media pari a 2.3 e massima pari a 6 elementi, mentre una hash table di 1 milione di bucket caricata con 2 milioni di elementi ha bucket con la stessa lunghezza media ma ha un massimo di ben 12 elementi. Per altri esempi e per il codice con cui sono stati calcolati questi risultati vedere [10].

Capitolo 5

Studio di una Hash Table ideale

In questa sezione studieremo in modo analitico le caratteristiche di una **hash table ideale**. Per maggiori informazioni, dimostrazioni e programmi per il calcolo dei dati visti di seguito può essere utile consultare [10].

Una funzione di hash ideale è una funzione che trasforma un dato in ingresso in una chiave hash con distribuzione di probabilità lineare. Consideriamo una hash table aperta con M bucket in cui vengono immessi N elementi diversi. Una funzione di hash lineare trasformerebbe ogni elemento in un valore di chiave hash compreso fra 0 e $(M-1)$ con probabilità $p = \frac{1}{M}$.

Sia ora q la probabilità del fallimento, cioè che un valore non sia trasformato in una data chiave hash:

$$q = 1 - p = \frac{M-1}{M}$$

La probabilità che, dopo avere immesso N elementi fra loro indipendenti, non ne sia capitato neanche uno in un bucket è data dalla probabilità congiunta di N fallimenti:

$$P(0) = q^N$$

La probabilità che un bucket sia stato occupato da almeno un elemento sarà

$$P(x > 1) = 1 - P(0)$$

per cui la percentuale di bucket occupati sarà

$$Q_{\%} = 100(1 - q^N)$$

Se la nostra funzione di hash è perfetta, ogni bucket avrà uguale probabilità $p = 1/M$ di ricevere un elemento, per cui il numero medio di elementi in ogni bucket sarà

$$E(x) = Np = \frac{N}{M}:$$

questo è il numero di confronti eseguiti in media per ogni ricerca nella hash table.

Se consideriamo però i soli bucket occupati, la lunghezza media delle code $E(x > 0)$ sarà data dalla distribuzione uniforme degli N valori nei $M(1 - P(0))$ bucket occupati:

$$E(x > 0) = \frac{N}{M(1 - q^N)}$$

Questa formula non dipende linearmente dal fattore di carico N/M e non può essere semplificata quando N è dello stesso ordine di grandezza o superiore a M.

La percentuale di bucket occupati e la lunghezza media dei bucket risentono quasi esclusivamente dal fattore di carico.

Esistono due modi di calcolare la lunghezza media delle code su ogni bucket, $E(x)$ e $E(x > l)$: se ho una hash table con 4 bucket, di cui uno vuoto, due con un elemento e uno con due, la lunghezza media dei bucket è

$$E(x) = \frac{1 \cdot 0 + 2 \cdot 1 + 1 \cdot 2}{4} = 1$$

ma la lunghezza media considerando i soli bucket occupati è:

$$E(x > 0) = \frac{2 \cdot 1 + 1 \cdot 2}{3} = 1.33$$

È quindi ovvio che la lunghezza media dei bucket occupati sia maggiore della lunghezza media di tutti i bucket.

Il numero medio di confronti necessari per ritrovare un elemento già presente nella hash table, quando i bucket non sono ordinati, è:

$$1 + \frac{E(x > 1) - 1}{2} = \frac{1 + E(x > 1)}{2}$$

Infatti si deve fare sempre almeno un confronto quando si trova l'elemento giusto oltre ai confronti con la metà degli elementi rimanenti in ogni bucket.

Se i bucket fossero ordinati, il numero medio di confronti per una ricerca con successo diventerebbe:

$$1 + \frac{E(x > 1) - 1}{4} = \frac{3 + E(x > 1)}{4}$$

La differenza sarebbe tanto misera da non compensare l'overhead necessario a mantenere ordinati i bucket. Invece usando come bucket delle liste self-organizing, cioè delle liste che spostano in testa gli elementi più ricercati l'overhead diminuisce ed il vantaggio, per hash table molto cariche, può essere notevole.

Data una hash table con M bucket e N elementi già inseriti, nell'ipotesi che la funzione di hash sia quella ideale e che valori immessi nella hash table siano equiprobabili, calcoliamo ora la probabilità che in un bucket ci siano k o più elementi.

Per prima cosa cerchiamo di capire qual è la probabilità $p(k)$ che ci siano esattamente k elementi in un bucket qualsiasi. Questa probabilità è data dalla somma delle probabilità di tutti i modi possibili in cui si sarebbe potuto creare un bucket di k elementi. Se k vale 1, questa probabilità è pari a N volte la probabilità che esattamente un solo valore su N sia capitato in quel bucket e tutti gli altri $N - 1$ siano finiti altrove. Si può ottenere un bucket lungo 1 se esso viene occupato dal primo elemento e da nessuno di quelli successivi, oppure dal secondo elemento e da nessuno degli altri, e così via. Si ha:

$$p(1) = Np(1-p)^{N-1}$$

dove p è sempre uguale a $1/M$, cioè alla probabilità di capitare in un certo bucket.

Passiamo ora a un k generico: la probabilità di avere un bucket di esattamente k elementi è equivalente al numero di modi diversi di ottenere esattamente k successi su N elementi. Tale probabilità è

$$p^k(1-p)^{N-k}$$

Il numero di combinazioni di k elementi presi in un gruppo di N è espresso matematicamente in questo modo:

$$\binom{N}{k} = \frac{N!}{k!(N-k)!}$$

quindi la probabilità di avere un bucket lungo esattamente k elementi è:

$$p(k) = \binom{N}{k} p^k(1-p)^{N-k}$$

Moltiplicando questa quantità per M (il numero di bucket nella tabella) si ottiene il numero medio di bucket con esattamente k elementi:

$$N(k) = M \cdot \binom{N}{k} p^k(1-p)^{N-k}$$

A questo punto basta cercare il massimo valore k_0 per cui il numero medio di bucket lungo k_0 sia superiore all'unità. In altre parole, sostituendo $N(k > k_0)$ con 1 si ottiene la massima lunghezza di un bucket per la quale sia statisticamente ragionevole prevedere almeno un esemplare

$$\max_{k_0} \binom{N}{k} p^k(1-p)^{N-k} \geq \frac{1}{M}$$

La successione formata dalle quantità

$$p(k) = \binom{N}{k} p^k(1-p)^{N-k}$$

per valori di $k = [0, 1, 2, \ldots]$ è ben nota agli studiosi di simulazione e di analisi delle prestazioni ed è chiamata distribuzione binomiale o distribuzione di Bernoulli. Dato un fenomeno con due possibili esiti, chiamati convenzionalmente successo e fallimento, fra i quali il successo ha probabilità p (e il fallimento, evidentemente, probabilità $q = 1 - p$), la distribuzione di Bernoulli fornisce, come abbiamo visto, la probabilità che si ottengano esattamente k successi su N eventi. Per $k = 0$ si ottiene $p(0) = (1-p)$ che è la probabilità del fallimento (nessun successo in N eventi). La somma:

$$p(k_0) = \sum_{k=1}^{k_0} \binom{N}{k} p^k(1-p)^{N-k}$$

per $k_0 = [1, 2, \ldots, N]$ fornisce la probabilità di non più di k_0 successi su N tentativi.

Capitolo 6

Implementazione in C++

Di seguito riporto il codice sorgente della mia implementazione della tabella hash.

6.1 Table.h

20 ⟨*Table.h* 20⟩≡

```
/*****************************************************
 *          TABELLE HASH CON LISTE DI TRABOCCO       *
 *    Implementazione di una Hash Table realizzata da  *
 *     ----------- Maria Luisa Guerriero ----------    *
 *****************************************************/

/* ------------------------------------------------------
* Header file contenente la dichiarazione della classe Table,
  che viene definita in Table.cpp
* Le chiavi all'interno della hash table sono uniche.
  L'inserimento di un elemento fallisce se la chiave e` gia`
  presente; inoltre si tiene conto del numero di volte in cui
  si tenta di inserire l'elemento.
* La hash table e` implementata usando i templates, per cui
  puo` contenere diversi tipi di dato.
* La dimensione della tabella aumenta e diminuisce
  dinamicamente. L'inserimento che causa l'ingrandimento e la
  cancellazione che causa il rimpicciolimento potrebbero
  impiegare molto tempo rispetto al normale
* L'ingrandimento si ha quando il numero di elementi e`
  maggiore del doppio della lunghezza della tabella, il
  rimpicciolimento quando il numero di elementi e` minore
  della meta` della lunghezza della tabella.

Le funzioni accessibili dall'esterno sono:

* Table - Costruttore: crea una hash table
* InitTable - Inizializza la hash table
```

```
* Insert - Inserisce un elemento nella hash table
* ShowTable - Mostra il contenuto della hash table
* Delete - Cancella un elemento dalla hash table
* Find - Cerca un elemento nella hash table

Le funzioni usate all’interno della classe sono:

* Hash - Calcola il valore hash dell’elemento da inserire
* ResizeTable - Ristruttura la hash table (se necessario viene
  chiamata da Insert e Delete)
* Insert - Inserisce un elemento (chiamata da ResizeTable)
* InsertionCount - Aggiorna il numero di tentati inserimenti di
  un elemento(chiamata da Insert se l’elemento e‘ gia‘ presente)
------------------------------------------------------- */

#ifndef TABLE_H
#define TABLE_H

#include "Lista.h"

//----------------------------------------------------------

template <class T> class Table {

public:

  int size; //dimensione tabella(numero bucket)
  int num_el; //numero elementi presenti
  Lista<T> * *hash_table; //puntatore alla hash table(che e‘
                          // composta da puntatori a liste)

  //----------------------------------------------------------
  /* ----- FUNZIONI PUBBLICHE ----- */

  /* Costruttore: crea una hash table */
  Table(int s = 13);

  /* Inizializza la hash table */
  void InitTable();

  /* Inserisce un elemento nella hash table
     INPUT: chiave da inserire */
  void Insert(T k);

  /* Mostra il contenuto della hash table */
  void ShowTable();

  /* Cancella un elemento dalla hash table
    INPUT: chiave da cancellare */
  void Delete(T k);
```

```
  /* Cerca un elemento nella hash table
     INPUT: chiave da cercare
     OUTPUT: true se e‘ presente, false altrimenti */
  bool Find(T k);

private:

  //-------------------------------------------------------------
  /* ----- FUNZIONI PRIVATE ----- */

  /* Calcola il valore hash dell'elemento da inserire
     INPUT: chiave da inserire
     OUTPUT: valore hash calcolato */
  int Hash(T k);

  /* Ristruttura la hash table (se necessario viene chiamata
     da Insert e Delete
     INPUT: indice della dimensione nel vettore m */
  void ResizeTable(int i);

  /* Inserisce un elemento (chiamata da ResizeTable)
     INPUT: chiave da inserire, num. di occorrenze della chiave,
            puntatore alla nuova tabella */
  void Insert(T k, int times, Lista<T> * *table);

  /* Aggiorna il numero di tentati inserimenti di un elemento
     INPUT: chiave gia‘ presente */
  void InsertionCount(T k);

};
//-------------------------------------------------------------

#include "Table.cpp"
#endif
```

6.2 Table.cpp

23 ⟨*Table.cpp* 23⟩≡

```
/*********************************************************
 *          TABELLE HASH CON LISTE DI TRABOCCO          *
 *    Implementazione di una Hash Table realizzata da   *
 *     ----------- Maria Luisa Guerriero ----------     *
 *********************************************************/
/* -----------------------------------------------------
* Source file contenente la definizione della classe Table,
  che viene dichiarata in Table.h
---------------------------------------------------- */

#ifndef TABLE_CPP
#define TABLE_CPP

#include <iostream.h>
#include <string>
#include <cstring>

#include "Table.h"

#define c 12

const int m[c] = {13,23,47,97,193,383,769,1531,3067,6143,12289,24571};

//------------------------------------------------------
/* ----- FUNZIONI PUBBLICHE ----- */

/* Costruttore: crea una hash table */
template <class T> Table<T>:: Table(int s)
{
  size = s;
  num_el = 0;
  hash_table = new Lista<T> * [size];
}
//------------------------------------------------------
/* Inizializza la hash table */
template <class T> void Table<T>:: InitTable()
{
  int i;
  for (i = 0; i < size; i++) hash_table[i] = new Lista<T>();
}
//------------------------------------------------------
/* Inserisce un elemento nella hash table
   INPUT: chiave da inserire */
template <class T> void Table<T>:: Insert(T k)
{
  int h, i=0;
  float a=(float)++num_el/size;
```

```
  if (Find(k))
    InsertionCount(k);
  else
  {
    if (a>2 && size <= m[c-2])
    {
      while(size >= m[i])
        i++;
      ResizeTable(i);
      a=(float)++num_el/size;
    }
    h = Hash(k);
    hash_table[h]->GoToFirst();
    hash_table[h]->Insert(k);
  }
}
//------------------------------------------------------------
/* Mostra il contenuto della hash table */
template <class T> void Table<T>:: ShowTable()
{
  int i;
  Lista<T> *l;

  for (i = 0; i < size; i++)
  {
    hash_table[i]->GoToFirst();
    if (!hash_table[i]->IsEndOfList())
    {
      cout << i << ":\n";
      l = hash_table[i];
      l->GoToFirst();
      while(!l->IsEndOfList())
      {
        cout << "    " << l->Read() << "\t(";
        cout << l->GetTimes() << ")" <<endl;
        l->Next();
      }
    }
  }
}
//------------------------------------------------------------
/* Cancella un elemento dalla hash table
   INPUT: chiave da cancellare */
template <class T> void Table<T>:: Delete(T k)
{
  int h, i=0;
  Lista<T> *l;
  float a=(float)--num_el/size;
```

```
  //trova la chiave da cancellare.
  h=Hash(k);

  l=hash_table[h];
  l->GoToFirst();

  while(!l->IsEndOfList() && l->Read()!= k)
  {
    l->Next();
  }
  if (l->IsEndOfList()) // scandita tutta la lista senza
                        // aver trovato l'elemento
    cout << k <<" non appartiene al dizionario" << endl;
  else  //l e` il nodo contenente la chiave da cancellare
  {
    l->Delete();
    cout << "cancellato " << k << endl;
  }

  if (a < 0.5 && size >= m[1])
  {
    while(size > m[i])
       i++;
    cout << a << "  " << size << endl;
    ResizeTable(i-1);
    a=(float)num_el/size;
  }
}
//-----------------------------------------------------------
/* Cerca un elemento nella hash table
   INPUT: chiave da cercare
   OUTPUT: true se e` presente, false altrimenti */
template <class T> bool Table<T>:: Find(T k)
{
  int i;
  Lista<T> *l;

  i=Hash(k);
  l = hash_table[i];
  l->GoToFirst();

  while(!l->IsEndOfList())
  {
    if (l->Read()==k)
      return true;
    l->Next();
  }

  return false;
}
```

```
//---------------------------------------------------------
/* ----- FUNZIONI PRIVATE ----- */

/* Calcola il valore hash dell'elemento da inserire
   INPUT: chiave da inserire
   OUTPUT: valore hash calcolato */

/* ---------------- Funzione hash generica ------------ */
template <class T> int Table<T>:: Hash(T k)
{
  long int h;

  h= (long) k;
  return (h % size);
}
/* ----------- Funzione hash di stringhe stile C ------- */
template <> int Table<char *>:: Hash(char * s)
{
  int len;
  long int h;

  if (len = strlen(s) == 1)
    h = (long) s[0]+s[0]*16+s[0]*256;
  else if (len = strlen(s) == 2)
    h = (long) s[0]+s[1]*16+s[1]*256;
  else
    h = (long) s[len/2]+s[len/2+1]*16+s[len-1]*256;

  return (h % size);
}
/* ---------- Funzione hash di stringhe stile C++ ------ */
template <> int Table<string>:: Hash(string s)
{
  int len;
  long int h;

  if (len = s.length() == 1)
    h = (long) s[0]+s[1]*16+s[0]*256;
  else if (len = s.length() == 2)
    h = (long) s[0]+s[1]*16+s[1]*256;
  else
    h = (long) s[len/2]+s[len/2+1]*16+s[len-1]*256;

  return (h % size);
}
//---------------------------------------------------------
/* Ristruttura la hash table (se necessario viene chiamata
   da Insert e Delete
   INPUT: indice della dimensione nel vettore m */
template <class T> void Table<T>:: ResizeTable(int i)
```

```
{
  int old_size = size;
  Lista<T> *l;
  Lista<T> * *new_table;

  num_el = 0;
  size = m[i];

  new_table = new Lista<T> * [size];

  for (int k = 0; k < size; k++) new_table[k] = new Lista<T>();

  for (int j = 0; j < old_size; j++)
  {
    hash_table[j]->GoToFirst();
    if (!hash_table[j]->IsEndOfList())
    {
      l = hash_table[j];
      l->GoToFirst();
      while(!l->IsEndOfList())
      {
        Insert(l->Read(), l->GetTimes(), new_table);
        l->Next();
      }
    }
  }
  delete [] hash_table;
  hash_table = new_table;
}
//----------------------------------------------------------
/* Inserisce un elemento (chiamata da ResizeTable)
   INPUT: chiave da inserire, num. di occorrenze della chiave,
          puntatore alla nuova tabella */
template <class T> void Table<T>::
         Insert(T k, int times, Lista<T> * *table)
{
  int h = Hash(k);

  num_el++;

  table[h]->GoToFirst();
  table[h]->Insert(k);
  table[h]->Previous();
  table[h]->SetTimes(times);
}
//----------------------------------------------------------
/* Aggiorna il numero di tentati inserimenti di un elemento
   INPUT: chiave gia‘ presente */
template <class T> void Table<T>:: InsertionCount(T k)
{
```

```
  int i;
  Lista<T> *l;

  i=Hash(k);
  l = hash_table[i];
  l->GoToFirst();

  while(!l->IsEndOfList())
  {
    if (l->Read()==k)
      l->SetTimes(l->GetTimes() + 1);
    l->Next();
  }
}
//-----------------------------------------------------------

#endif
```

6.3 Lista.h

29 ⟨*Lista.h* 29⟩≡

```
/*****************************************************
 *          LISTA BIDIREZIONALE CON SENTINELLA          *
 *      Implementazione di una Lista realizzata da      *
 *     ----------- Maria Luisa Guerriero ----------     *
 *****************************************************/

/* ----------------------------------------------------
* Header file contenente la dichiarazione della classe Lista,
  che viene definita in Lista.cpp
* Basata in parte su quella di Battiti e Brunato, ed in parte
  su quella di Villani.
* La lista e` implementata usando i templates, per cui puo`
  contenere diversi tipi di dato.
* La lista usa due puntatori, uno al nodo precedente e l'altro
  al nodo successivo; un nodo, detto sentinella, indica la fine
  della lista (lista bidirezionale con sentinella).

Le funzioni accessibili dall'esterno sono:

* Lista - Costruttore: crea una lista
* ~Lista - Distruttore: cancella la lista rilasciando la memoria
* Next - Sposta il nodo corrente sul nodo successivo
* Previous - Sposta il nodo corrente sul nodo precedente
* GoToFirst - Sposta il nodo corrente sul primo nodo della lista
* GoToEndOfList - Sposta il nodo corrente oltre l'ultimo nodo
  della lista
* IsEmpty - Controlla se la lista e` vuota
* IsFirst - Controlla se il nodo corrente e` sul primo nodo
* IsEndOfList - Controlla se il nodo corrente e` oltre l'ultimo nodo
* Insert - Inserisce un nodo nella posizione corrente
* Write - Cambia la chiave del nodo corrente
* Read - Legge la chiave del nodo corrente
* Delete - Cancella il nodo alla posizione corrente
* GetTimes - Ritorna il numero di tentati inserimenti del nodo alla
  posizione corrente
* SetTimes - Aggiorna il numero di tentati inserimenti del nodo alla
  posizione corrente
* GetLenght - Ritorna la lunghezza (numero di elementi) della lista
---------------------------------------------------- */

#ifndef LISTA_H
#define LISTA_H

//----------------------------------------------------

/* -- Classe base per rappresentare un nodo della Lista -- */
class nodoBase {
```

```
public:
  int times; // numero di tentati inserimenti della chiave
  nodoBase * next; // puntatore al nodo successivo
  nodoBase * prev; // puntatore al nodo precedente
};

/* -------------------- Classe nodo --------------------- */
template <class T> class nodo: public nodoBase {

public:

  T key; // chiave dell'elemento contenuto nel nodo

  nodo(const T& k): // Copy constructor
    key(k)
    {}
};
//-----------------------------------------------------------

/* --------- Classe per rappresentare la Lista ---------- */
template <class T> class Lista {

  nodoBase * sentinel; // puntatore alla sentinella
  nodoBase * current; // puntatore al nodo corrente
  int len; // lunghezza della lista

public:

  //-----------------------------------------------------------
  /* ----- FUNZIONI PUBBLICHE ----- */

  /* Costruttore: crea una lista */
  Lista();

  /* Distruttore: cancella la lista rilasciando la memoria */
  ~Lista();

  /* Sposta il nodo corrente sul nodo successivo */
  void Next() { current = current->next; }

  /* Sposta il nodo corrente sul nodo precedente */
  void Previous() { current = current->prev; }

  /* Sposta il nodo corrente sul primo nodo della lista */
  void GoToFirst() { current = sentinel->next; }

  /* Sposta il nodo corrente oltre l'ultimo nodo della lista */
  void GoToEndOfList() { current = sentinel; }
```

```
  /* Controlla se la lista e‘ vuota
     OUTPUT: 1 se e’ vuota, 0 altrimenti */
  int IsEmpty() { return (sentinel == sentinel->next); }

  /* Controlla se il nodo corrente e‘ sul primo nodo
     OUTPUT: 1 se e‘ sul primo nodo, 0 altrimenti */
  int IsFirst() const { return (current == sentinel->next); }

  /* Controlla se il nodo corrente e‘ oltre l’ultimo nodo
     OUTPUT: 1 se e‘ oltre l’ultimo nodo, 0 altrimenti */
  int IsEndOfList() const { return (current == sentinel); }

  /* Inserisce un nodo nella posizione corrente
     INPUT: chiave del nodo da inserire */
  void Insert(T s);

  /* Cambia la chiave del nodo corrente
     INPUT: stringa da inserire come nuova chiave */
  void Write(T s) { current->key = s; }

  /* Legge la chiave del nodo corrente
     OUTPUT: stringa letta */
  T Read()
  { nodo<T>* nptr = (nodo<T>*) Lista::current;
    return nptr->key; }

  /* Cancella il nodo alla posizione corrente
     OUTPUT: chiave dell’elemento cancellato */
  T Delete();

  /* Ritorna il numero di tentati inserimenti del nodo
     alla posizione corrente
     OUTPUT: times = numero di tentati inserimenti */
  int GetTimes()
  { nodo<T>* nptr = (nodo<T>*) Lista::current;
    return nptr->times; }

  /* Aggiorna il numero di tentati inserimenti del nodo
     alla posizione corrente
     INPUT: times = numero di tentati inserimenti */
  void SetTimes(int times) { current->times=times; }

  /* Ritorna la lunghezza (numero di elementi) della lista
     OUTPUT: len = lunghezza della lista */
  int GetLenght() { return len; }
};
//-----------------------------------------------------------

#include "Lista.cpp"
#endif
```

6.4 Lista.cpp

32 ⟨*Lista.cpp* 32⟩≡

```
/*****************************************************
 *          LISTA BIDIREZIONALE CON SENTINELLA         *
 *      Implementazione di una Lista realizzata da     *
 *     ----------- Maria Luisa Guerriero ----------    *
 *****************************************************/
/* -----------------------------------------------------
* Source file contenente la definizione della classe Lista,
  che viene dichiarata in Lista.h
----------------------------------------------------- */

#ifndef LISTA_CPP
#define LISTA_CPP
#include "Lista.h"

//-----------------------------------------------------
/* ----- FUNZIONI PUBBLICHE ----- */

/* Costruttore: crea una lista */
template <class T> Lista<T>:: Lista()
{
  sentinel = new nodoBase;   // Creo il nodo sentinella
  sentinel->next = sentinel; // Inizializzo la sentinella
  sentinel->prev = sentinel; //          "
  current = sentinel;
  len=0;
}
//-----------------------------------------------------
/* Distruttore: cancella la lista rilasciando la memoria */
template <class T> Lista<T>:: ~Lista()
{
  GoToFirst();          // Vado al primo elemento della Lista
  while(!IsEndOfList()) // Finche` non arrivo alla sentinella
    Delete();           // Cancello un elemento
  delete sentinel;      // Elimino infine anche la sentinella
}
//-----------------------------------------------------
/* Inserisce un nodo nella posizione corrente
   INPUT: chiave del nodo da inserire */
template <class T> void Lista<T>::Insert(T s)
{

  nodo<T>* newnodo = new nodo<T>(s); // Costruiamo un nuovo nodo

  newnodo->next = current;
          // Il nuovo nodo e` inserito PRIMA del nodo corrente
  newnodo->prev = current->prev; //              "
  newnodo->key = s;  // Inizializzo il valore del nodo
```

```
  current->prev = newnodo;
  newnodo->prev->next = newnodo;
  newnodo->times=1;
  len++;
}
//-----------------------------------------------------------
/* Cancella il nodo alla posizione corrente
   OUTPUT: chiave dell'elemento cancellato */
template <class T> T Lista<T>:: Delete()
{
  nodo<T>* nptr = (nodo<T>*) Lista::current;

  T tmp = nptr->key;
  nodo<T>* tmpp = (nodo<T>*) current->next;

  current->next->prev = current->prev;
  current->prev->next = current->next;
  delete current;
  len--;
  current = tmpp;
  return tmp;
}
//-----------------------------------------------------------

#endif
```

Ho inserito anche due file che possono essere utili per capire come va usata la tabella hash. Un terzo file (CountWords.cpp) lo abbiamo già visto nella sezione 1.2.

6.5 TestDizionario.cpp

34 ⟨*TestDizionario.cpp* 34⟩≡

```
/************************************************************
 *             TABELLE HASH CON LISTE DI TRABOCCO            *
 *File di test e di esempio d'uso del dizionario realizzato da*
 *       ----------- Maria Luisa Guerriero ----------        *
 ************************************************************/
/* -----------------------------------------------------
* In questo esempio il dizionario e` utilizzato per contenere
  delle stringhe (char *), ma si potrebbe usare anche per
  inserire altri tipi di dato (ad esempio int, float,...).
---------------------------------------------------- */

#include <iostream.h>
#include <string>

#include "Table.h"

//-----------------------------------------------------

void main()
{
 cout<<"Creo una hash table per contenere stringhe(char *)\n";
 Table<char *> T;
 char * str;

 T.InitTable();

 cout<<"Inserisco nell'ordine le stringhe: ";
 cout <<"\"questa\" \"e`\" \"una\" \"prova\"\n";

 str="questa"; T.Insert(str);
 str="e`"; T.Insert(str);
 str="una"; T.Insert(str);
 str="prova"; T.Insert(str);

 cout<<"Mostro il contenuto della hash table\n";
 T.ShowTable();

 cout<<"Inserisco la stringa \"prova\", che e` gia` ";
 cout <<"presente nella hash table...\n";
 str="prova";
 T.Insert(str);
```

```
 cout<<"...e verifico che non venga inserita nuovamente\n";
 T.ShowTable();

 cout<<"Cancello la stringa \"questa\"...\n";
 T.Delete("questa");

 cout<<"...controllo il contenuto della hash table\n";
 T.ShowTable();

 cout<<"Cancello le altre stringhe inserite\n";
 T.Delete("e‘");
 T.Delete("una");
 T.Delete("prova");

 cout<<"La tabella ora e‘ vuota\n";
 T.ShowTable();
}
//-----------------------------------------------------------
```

6.6 TestInterattivoDizionario.cpp

36 ⟨*TestInterattivoDizionario.cpp* 36⟩≡

```
/*****************************************************
 *         TABELLE HASH CON LISTE DI TRABOCCO        *
 *File per utilizzo guidato del dizionario realizzato da*
 *     ----------- Maria Luisa Guerriero ----------     *
 *****************************************************/
/* -----------------------------------------------------
* In questo esempio il dizionario e‘ utilizzato per contenere
  delle stringhe (string), ma si potrebbe usare anche per
  inserire altri tipi di dato (ad esempio int, float,...).
----------------------------------------------------- */

#include <iostream.h>
#include <string>
#include <fstream.h>

#include "Table.h"

#define STRLEN 80

//-----------------------------------------------------

int main(void)
{
  Table<string> T;
  string str;
  char sc;
  char filename[STRLEN];
  int i=0;

  cout <<"In questo esempio il dizionario e‘ utilizzato ";
  cout <<"per contenere delle stringhe (string)\n";

  ifstream in;

  T.InitTable();

  while(1)
  {
    cout << "1) Inserire un elemento\n";
    cout << "2) Inserire piu‘ elementi da file\n";
    cout << "3) Inserire piu‘ elementi da tastiera\n";
    cout << "4) Visualizzare il dizionario\n";
    cout << "5) Cancellare un elemento\n";
    cout << "6) Cercare un elemento\n";
    cout << "7) Uscire\n";

    cin >> sc;
```

```
int esc=0;
switch (sc)
{
  /* 1) Inserire un elemento */
  case '1': cin >>str;
            T.Insert(str);
            break;

  /* 2) Inserire piu` elementi da file */
  case '2': cout << "Nome file: ";
            cin >> filename;
            in.open(filename);
            while(!in)
            {
              cout<<"\nImpossibile aprire il file.\n";
              cout<<"Scrivere esc per tornare al menu.\n";
              cin>>filename;

              if(strcmp(filename, "esc")==0)
        {
          esc=1;
          break;
        }
              in.close();
              in.open(filename);
            }
            if (esc==1) break;

            while (!in.eof())
            {
              char buffer[STRLEN];

              in.getline(buffer, STRLEN-1);
              if (buffer==NULL) continue;
              str=buffer;

              const char* s1=str.c_str();
              if (*s1 =='\0') continue;
              T.Insert(str);
            }
            in.close();
            break;

  /* 3) Inserire piu` elementi da tastiera */
  case '3': const char* s1;
            cout<<"Inserire gli elementi. ";
            cout <<"Digitare * per interrompere.\n";
            do
            {
              cin >>str;
```

```
                s1=str.c_str();
                if (*s1 != '*')
                  T.Insert(str);
              }
              while (*s1 != '*');
              break;

      /* 4) Visualizzare il dizionario */
      case '4': T.ShowTable();
              break;

      /* 5) Cancellare un elemento */
      case '5': cin >>str;
              T.Delete(str);
              break;

      /* 6) Cercare un elemento */
      case '6': cin >>str;
              if (T.Find(str))
                cout << "trovata  la chiave \"" << str << "\"\n";
              else
              {
                cout << "la chiave \"" <<  str;
                cout << "\" non appartiene al dizionario\n";
              }
              break;

      /* 7) Uscire */
      case '7': exit(0);
    }
  }
}
//-----------------------------------------------------------
```

Parte II

HEAP BINOMIALI

Capitolo 7

Introduzione

L'utilizzo che viene fatto degli **heap** in genere è quello di realizzare **code con priorità**, in cui gli elementi inseriti vengono ordinati in base alla loro chiave (che rappresenta la priorità) e l'elemento che viene estratto di volta in volta è quello con priorità più alta (cioè chiave più bassa).

Uno heap binomiale è una collezione di **alberi binomiali** che rispettano l'ordinamento degli heap, quindi per approfondire l'argomento bisogna innanzitutto definire sia l'heap binario che l'albero binomiale e mostrarne alcune proprietà. In seguito vedremo le proprietà degli heap binomiali, la loro rappresentazione, e le operazioni supportate da queste strutture. Infine mostrerò la mia implementazione di uno heap binomiale.

Gli heap binomiali sono strutture dati tramite cui, oltre alle classiche operazioni di inserimento e cancellazione, ricerca ed estrazione dell'elemento minimo, si possono realizzare facilmente ed in modo efficiente altre operazioni: la loro importanza è dovuta soprattutto al modo efficiente con cui due heap possono essere uniti. Se si vogliono unire due heap binari, ognuno memorizzato in un array, l'unico modo è concatenare i due array e costruire uno heap composto. Se i due heap contengono N ed M elementi, il costo nel caso pessimo con gli heap binari è $\mathrm{O}(N+M)$, mentre con gli heap binomiali è $\mathrm{O}(\log N)$

Inoltre gli heap binomiali supportano l'operazione di *DecreaseKey* molto più facilmente degli heap binari. Anche gli heap binari standard possono supportare la *DecreaseKey*, ma richiedono una qualche struttura extra (ad esempio una lista o un albero binario di ricerca). In pratica questa funzione serve se le priorità degli elementi inseriti nella coda possono aumentare nel corso del tempo: ad esempio, per evitare che un elemento a bassa priorità resti per sempre nella coda, se ne può aumentare la priorità ad intervalli di tempo regolari.

Vedremo come usando gli heap binomiali tutte queste operazioni possano essere realizzate in $\mathrm{O}(\log N)$.

Gli heap binari funzionano bene per tutte le operazioni supportate dagli heap binomiali, a parte la *Union*. Se deve essere realizzata anche questa funzione conviene utilizzare delle strutture più sofisticate, come appunto gli heap binomiali. Esiste comunque una struttura dati che permette di implementare in modo ancor più efficiente questa funzione: lo heap di Fibonacci, con cui l'operazione di unione (così come quella di creazione, inserzione, ricerca del minimo e *DecreaseKey*) richiede un tempo costante (in un'analisi ammortizzata).

Per quanto riguarda l'operazione di ricerca di un elemento tutte le strutture basate sugli heap (sia quelli binari, sia quelli binomiali, sia quelli di Fibonacci) sono inefficienti.

7.1 Operazioni sugli heap binomiali

Vediamo quali sono le operazioni supportate dagli heap binomiali:

- **MakeHeap** - Crea e ritorna un nuovo heap binomiale vuoto
- **Union** - Unisce due heap binomiali dello stesso ordine, creando e ritornando un nuovo heap che contiene tutti i nodi degli heap iniziali, i quali vengono "distrutti" da questa operazione
- **Insert** - Inserisce un elemento nello heap binomiale
  È un caso speciale della *Union*, dove uno dei due heap da combinare è un elemento singolo
- **Minimum** - Trova l'elemento con chiave minima nello heap binomiale e ritorna un puntatore al nodo che lo contiene
- **ExtractMin** - Cancella l'elemento con chiave minima dallo heap, ritornando un puntatore al nodo eliminato
- **DecreaseKey** - Decrementa la chiave di un elemento dello heap binomiale, aumentandone quindi la priorità
- **Delete** - Cancella un elemento dallo heap binomiale
- **IsEmpty** - Controlla se lo heap binomiale è vuoto
- **Size** - Ritorna il numero di elementi dello heap binomiale

Operazioni	Heap binario	Heap binomiale
MakeHeap	$\Theta(1)$	$\Theta(1)$
Union	$\Theta(N)$	$O(\log N)$
Insert	$\Theta(\log N)$	$O(\log N)$
Minimum	$\Theta(1)$	$O(\log N)$
ExtractMin	$\Theta(\log N)$	$\Theta(\log N)$
DecreaseKey	$\Theta(\log N)$	$\Theta(\log N)$
Delete	$\Theta(\log N)$	$\Theta(\log N)$
IsEmpty	$\Theta(1)$	$\Theta(1)$
Size	$\Theta(1)$	$\Theta(1)$

Tabella 7.1: Riassunto dei costi nel caso pessimo per quanto riguarda gli heap binari e quelli binomiali (N è il numero di elementi presenti nello heap)

Nel Capitolo 9 le passeremo in rassegna tutte, analizzandone la complessità. Prima però deve essere chiaro cosa sia e come venga realizzato uno heap binomiale.

7.2 Alberi Binomiali

Un albero binomiale B_k (di **ordine** k) è un albero ordinato definito ricorsivamente come mostrato in Figura 7.1:

- l'albero binomiale B_0 è formato da un solo nodo
- l'albero binomiale B_k è formato da due alberi binomiali B_{k-1} collegati insieme in modo che la radice di uno sia il figlio più a sinistra della radice dell'altro.

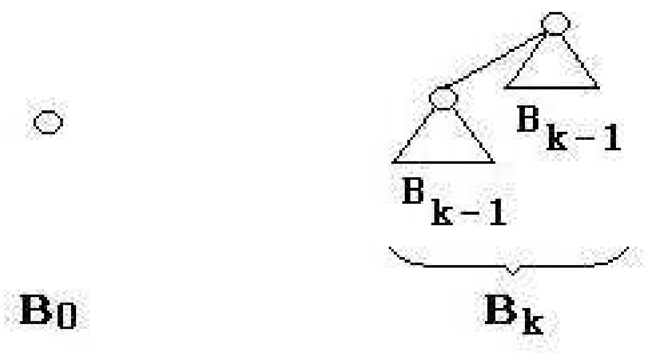

Figura 7.1: Albero binomiale: definizione ricorsiva

La Figura 7.2 mostra gli alberi binomiali da B_0 a B_4.

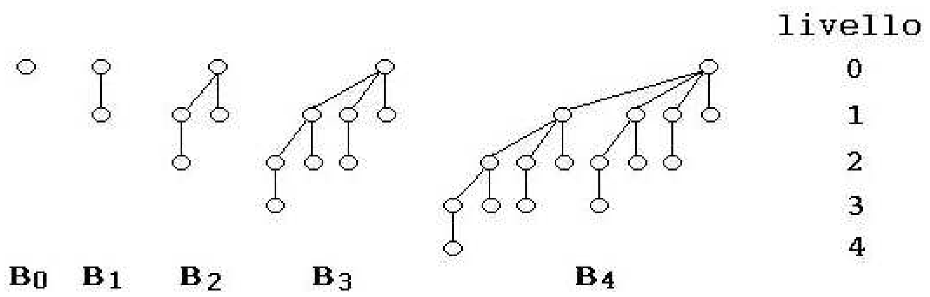

Figura 7.2: Alberi binomiali da B_0 a B_4

Ecco le proprietà che caratterizzano un albero binomiale B_k:

1. è formato da 2^k nodi: infatti dalla definizione di sopra si ricava che se N_k è il numero di nodi di B_k, allora $N_{k+1} = 2N_k$, e, dato che $N_0 = 1$, allora $N_k = 2^k$, per $k \geq 0$.
2. l'altezza dell'albero è k: infatti l'altezza H_k di un albero binomiale B_k è $1 + H_{k-1}$, e, dato che $H_0 = 0$, risulta che $H_k = k$, per $k \geq 0$.
3. ci sono esattamente $\binom{k}{i} = k!/i!(k-i)!$ nodi al livello i, per $0 \leq i \leq k$.
   Da questa proprità deriva il nome "albero binomiale", siccome il termine $\binom{k}{i}$ è il **coefficiente binomiale**.

4. la radice dell'albero ha **grado** k (cioè ha k figli), che è il grado massimo tra tutti i nodi.

5. se i figli della radice vengono numerati da sinistra verso destra con $k, k-1, \ldots, 1$, l'i-esimo figlio è la radice di un sottoalbero B_{i-1}. In altre parole l'albero consiste di una copia di ognuno di $B_0, B_1, B_2, \ldots, B_{k-1}$, tutti connessi con le loro radici alla radice di B_k.

Combinando le proprietà 1 e 4 risulta che il grado massimo di ogni nodo in un albero binomiale con N nodi è $\log N$.

7.3 Heap

Altra nozione necessaria per comprendere gli heap binomiali è quella di heap. Questa è una struttura usata per realizzare le code con priorità, cioè tipi particolari di insieme sui cui elementi è definita una relazione di ordinamento totale "$\leq$". In una coda con priorità si possono inserire elementi e si può estrarre solo quello minimo.

Uno heap può essere rappresentato come un albero (Figura 7.3) che ha come radice l'elemento minimo e su cui è facile definire la proprietà fondamentale che caratterizza gli heap in questo modo: ogni nodo contiene un elemento maggiore o uguale di quello contenuto nel padre.

Esistono più tipi di heap: il più semplice è lo heap binario, in cui gli elementi vengono memorizzati in un vettore, come raffigurato in Figura 7.3; esistono poi gli heap binomiali, che vedremo dettagliatamente d'ora in avanti, e gli heap di Fibonacci.

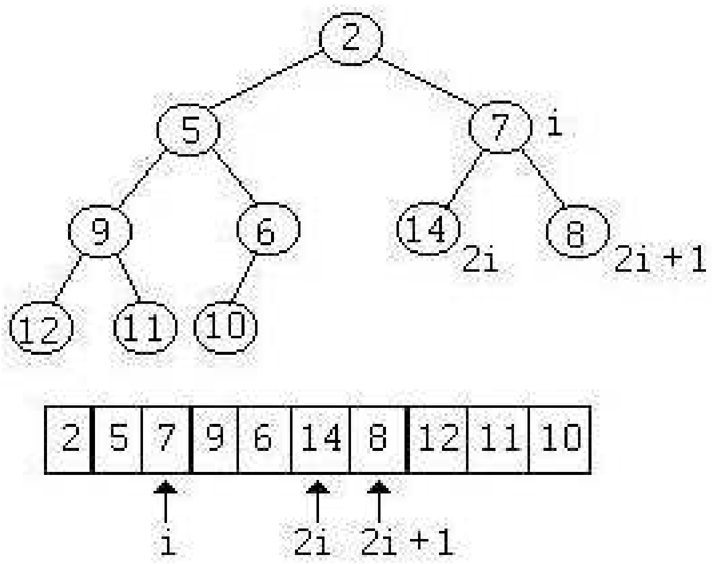

Figura 7.3: Esempio di heap

Capitolo 8

Heap Binomiali

8.1 Definizione e proprietà

Una **coda binomiale** è una foresta di alberi binomiali, in cui può essere presente al più un albero per ogni indice (ossia la cui radice ha un dato grado). Ad esempio la Figura 7.2 mostra una coda binomiale contenente 5 alberi binomiali, con $1 + 2 + 4 + 8 + 16 = 31$ nodi.

Se ci sono delle etichette (chiavi) associate ai nodi di un albero binomiale, ed imponiamo un ordinamento in cui la chiave di un nodo è minore o uguale delle chiavi dei figli, possiamo costruire uno heap con una coda binomiale, ottenendo perciò una struttura chiamata heap binomiale.

Abbiamo visto che due alberi binomiali dello stesso ordine k possono essere fusi insieme ottenendo un albero di ordine $k + 1$, semplicemente attaccando la radice di uno come figlio della radice dall'altro. Con l'ordinamento imposto sappiamo che gli alberi da combinare rispettano la proprietà degli heap, perciò la chiave minore è nella radice dell'albero; sappiamo allora che la radice dell'albero risultante deve essere la minore delle radici dei due alberi da unire. Dunque per combinare i due alberi semplicemente attacchiamo quello con la radice con chiave maggiore come figlio della radice dell'altro.

Ad esempio la Figura 8.1 illustra come due alberi binomiali heap-ordinati di ordine 2 vengono combinati insieme in un albero binomiale heap-ordinato di ordine 3.

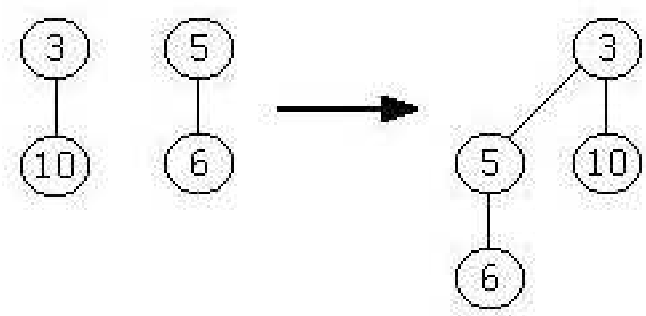

Figura 8.1: Unione di due alberi binomiali

Formalizzando, uno **heap binomiale** H è un insieme di alberi binomiali che soddisfa le seguenti proprietà:

1. ogni albero binomiale di H rispetta l'ordinamento heap, cioè la chiave di un nodo deve essere maggiore o uguale alla chiave del padre

2. non ci può essere più di un albero binomiale in H la cui radice ha un dato grado

Queste proprietà implicano che:

- la radice di un albero heap-ordinato contiene la chiave più piccola dell'albero

- uno heap binomiale H di N nodi è composto al più da $\lfloor \log N \rfloor + 1$ alberi binomiali, ed i gradi delle radici degli alberi sono un sottoinsieme di $0, 1, ..., \lfloor \log N \rfloor$.

La prima osservazione è ovvia; cerchiamo invece di capire il perché della seconda: sappiamo che la rappresentazione binaria di N ha $\lfloor \log N \rfloor + 1$ bit $(b_{\lfloor \log N \rfloor}, b_{\lfloor \log N \rfloor - 1}, \ldots, b_0)$, così che $N = \sum_{i=0}^{\lfloor \log N \rfloor} b_i 2^i$. Perciò, per le Proprietà 1 e 2 degli alberi binomiali, la struttura dello heap binomiale H è legata alla rappresentazione binaria di N in questo modo: il bit $b_i = 1$ se e solo se B_i compare in H. Quindi H contiene al massimo $\lfloor \log N \rfloor + 1$ alberi binomiali. Inoltre ne consegue che i gradi delle radici degli alberi sono un sottoinsieme di $0, 1, ..., \lfloor \log N \rfloor$.

Vediamo un esempio: la Figura 8.2 mostra uno heap binomiale H con 11 nodi. Sappiamo che la rappresentazione di 11 in binario è (1011): H è infatti composto dagli alberi binomiali heap-ordinati B_3, B_1, e B_0, che hanno 8, 2, e 1 nodi rispettivamente, per un totale di 11 nodi.

Notare che la rappresentazione di uno heap con N nodi è unica, poiché la rappresentazione binaria di N è unica. Nel nostro esempio dunque l'unica possibilità era proprio avere gli alberi B_3, B_1, e B_0.

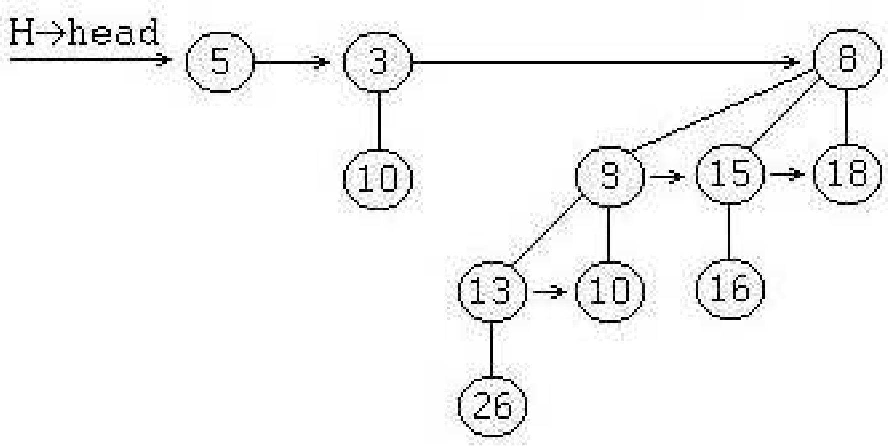

Figura 8.2: Esempio di heap binomiale con B_0, B_1, e B_3

8.2 Un esempio

Vediamo per esempio come costruire un albero che contenga le chiavi:
C O M P U T E R S.

```
-->   C      (il primo elemento viene semplicemente inserito)

-->   C    (abbiamo fuso 2 alberi B0 in un albero B1, mantenendo
      |     la proprieta' degli heap confrontando le 2 chiavi e
      O     usando la maggiore come radice del nuovo albero)


-->   M   C    (ora abbiamo un albero B0 ed un albero B1)
          |
          O

-->  M   C  -->   C  (abbiamo fuso 2 alberi B0 in un albero B1,
     |   |       /|   ma poi abbiamo unito i 2 alberi B1 perche'
     P   O      M O   non possono essercene due, attaccando la
                |     radice con la chiave piu' grande come
                P     nuovo figlio di quella con chiave minore)

-->  U   C    (in questo caso non e' necessaria nessuna unione)
        /|
       M O
       |
       P

-->  T     C    (ora abbiamo fuso 2 alberi B0 in un albero B1)
     |    / \
     U   M   O
         |
         P

-->  E   T    C      (abbiamo un albero B0, un B1 ed un B2)
         |   / \
         U  M   O
            |
            P

-->  E   T    C     -->     E      C      -->      C
     |   |   / \           / \    / \            / | \
     R   U  M   O         T   R  M   O           E  M  O
            |             |      |              /|  |
            P             U      P             T R  P
                                                |
                                                U
```

(nell'inserire la chiave R nello heap binomiale sono avvenute diverse unioni: R ed E sono state fuse in un albero binomiale di ordine 1, ma poiche' c'era gia' un B1 (con radice T), questi due alberi sono stati fusi producendo un B2, che era gia'

```
presente (con radice C), cosi‘ questi due alberi sono stati
uniti formando un albero binomiale di ordine 3)

-->  S       C      (nell’ultima inserzione e’ bastato
           / | \     inserire un albero B0)
         E   M  O
        / \  |
       T   R P
       |
       U
```

Dello heap binomiale che otteniamo alla fine fanno parte due alberi binomiali: uno di ordine 0 e l’altro di ordine 3.

Capitolo 9

Rappresentazione e analisi delle operazioni

Per rappresentare uno heap binomiale per prima cosa è necessario mantenere una lista che contenga le radici degli alberi binomiali (e quindi gli alberi stessi) che ne fanno parte; le radici sono inserite nella lista in modo che i loro gradi siano in ordine strettamente crescente.

Allo heap binomiale `H` si accede attraverso `Head`, un puntatore al primo elemento della lista delle radici (non a quella con chiave minore) di `H`.

Nel rappresentare uno heap binomiale, ad ogni nodo `x` vengono associati:

1. una chiave (`key`) e gli altri eventuali dati necessari
2. un puntatore (`parent`) al padre. Se `x` è una radice, è settato a NULL
3. un puntatore (`child`) al figlio più a sinistra. Se `x` non ha figli, `child` è settato a NULL
4. un puntatore (`sibling`) al fratello alla sua destra o, se `x` è una radice, al nodo successivo ad `x` nella lista delle radici. Se nessun nodo segue `x`, `sibling` è settato a NULL
5. un intero (`degree`) contenente il grado, cioè il numero di figli di `x`

I punti 3 e 4 insieme dicono che i figli di un nodo sono tenuti in una lista, ed ogni nodo ha il puntatore al primo figlio (`child`); per accedere agli altri si deve scandire la lista seguendo i campi `sibling` di ogni nodo figlio.

Un'altra possibilità è che ogni nodo abbia un puntatore al sibling destro e uno a quello sinistro.

Vediamo ora di passare in rassegna le operazioni, già accennate nel Capitolo 7.1, che si possono effettuare sugli heap binomiali, analizzandone la complessità.

- **MakeHeap**:
  Il puntatore `Head` viene allocato e settato a NULL. L'operazione costa dunque O(1).
- **Union**:
  Come già accennato questa operazione è molto efficiente usando gli heap

binomiali, poiché può essere realizzata in $O(\log N)$. I passi da compiere sono:

1. Fare una nuova collezione consistente nell'unione delle collezioni dei due heaps.
2. Se la nuova collezione contiene due alberi dello stesso grado, combinarli ponendo la radice con la chiave minore come radice del nuovo albero. Nell'unire gli alberi si procede dai più piccoli ai più grandi, in modo analogo alla somma di 2 numeri binari. Su [4] ci sono alcuni disegni che aiutano a capire il funzionamento dell'operazione.

Unire due alberi dello stesso ordine costa tempo costante (basta cambiare un puntatore); ci sono $O(\log N)$ alberi, quindi combinare 2 heap costa $O(\log N)$.

Nell'implementazione fornita il costo di questa operazione in un'analisi ammortizzata è addirittura $O(1)$.

- **Insert**:
  Il costo nel caso pessimo è $O(\log N)$, ma si può dimostrare che con gli heap binomiali il costo ammortizzato per inserzione è $O(1)$!
- **Minimum**:
  Una possibilità è quella di mantenere un puntatore al nodo con la chiave minore (che è sicuramente la radice di uno degli alberi della foresta), aggiornandolo eventualmente dopo l'inserimento di una nuova chiave; oppure si può semplicemente scandire la lista delle radici e cercare quella con la chiave più piccola (ricerca che si può effettuare in $O(\log N)$ poiché ci sono al più $\lfloor \log N \rfloor + 1$ alberi).
- **ExtractMin**:
  Per usare la coda binomiale come heap deve essere possibile, ovviamente, cancellare la chiave più piccola mantenendo le proprietà dello heap.

  Dopo aver cancellato il nodo con chiave minima, trovato come descritto nella *Minimum*, resta una foresta dei suoi figli, che deve essere unita a quella che comprende gli altri alberi binomiali dello heap.

  Ad esempio consideriamo la cancellazione della chiave A dallo heap costruito in precedenza. Dopo aver eliminato la radice contenente A dobbiamo unire due foreste, una che consiste dell'albero contenente M ed S, e l'altra che è composta dagli alberi radicati in L G ed I. Bisogna perciò unire i due alberi B_1 con radici M e G, e poi unire l'albero B_2 risultante con quello con radice I. Il risultato finale è una foresta composta da 2 alberi (B_0 e B_3), in cui il puntatore al nodo con chiave massima punta a G.

```
S      O  M    E      -->                 E
          |   /|                        / | \
          P  T R                      M   T  R
             |                       / \  |
             U                      O   P U
                                    |
                                    S
```

In generale le operazioni da effettuare sono:

1. Cancellare dalla lista delle radici quella con chiave minima. `Head`, che puntava al minimo, ora va a puntare temporaneamente a qualche altra radice: $O(1)$ oppure $O(\log N)$ a seconda della realizzazione della *Minimum*
2. A causa della cancellazione della radice, uno dei B_i è stato diviso in sottoalberi, che vanno ricombinati con gli altri alberi dello heap, che sono rimasti immutati. Bisogna quindi unire gli alberi che hanno lo stesso grado, fino ad ottenere al massimo un albero per ogni grado: $O(\log N)$
3. Collegare le radici del nuovo insieme di alberi nella lista e far puntare `Head` alla nuova radice minima: $O(\log N)$

La cancellazione del nodo con chiave minima impiega perciò $\Theta(\log N)$.

- **DecreaseKey**:
  Per permettere questa operazione l'utente riceve un ticket quando un oggetto è inserito per la prima volta nello heap. Nella rappresentazione con code binomiali all'utente ritorna un puntatore al record dell'oggetto appena inserito. Per decrementare il valore della chiave, l'utente passa il ticket e la nuova priorità, e la *DecreaseKey* porta l'elemento in su finché viene ripristinato l'ordinamento heap. Si può realizzare anche la *IncreaseKey*, che sposta il nodo in giù fino a quando la chiave è minore di quelle dei figli. Nell'implementare questa operazione, per evitare di dover cercare il figlio di un nodo con chiave minore (per confrontarla con la nuova priorità del padre ed eventualmente scambiarli) si può aggiungere ad ogni nodo un puntatore che indica il figlio con la chiave maggiore, che deve essere aggiornato ad ogni inserzione, cancellazione, *IncreaseKey* e *DecreaseKey*. Notare invece che il ticket di un elemento non cambia mai: è l'indirizzo del nodo che contiene le informazioni dell'elemento (dato e priorità).

  La mia implementazione si basa su quella in [4] e non utilizza questo sistema del ticket, per cui passare alla funzione il puntatore al nodo desiderato richiederebbe la ricerca del nodo all'interno dello heap, che come detto è una operazione molto inefficiente con questa struttura dati.

  Escludendo il tempo che servirebbe per la ricerca del nodo, l'operazione di *DecreaseKey* impiega $\Theta(\log N)$ tempo, perché la massima profondità di un nodo è $\lfloor \log N \rfloor$.

- **Delete**:
  Per questa operazione vale lo stesso discorso fatto per la *DecreaseKey*: la *Delete* impiega $\Theta(\log N)$ tempo, ma bisogna passare alla funzione il puntatore al nodo da cancellare.

- **IsEmpty**:
  Questa funzione booleana impiega $\Theta(1)$ tempo (ritorna vero se la testa della lista delle radici è NULL).

- **Size**:
  Costa $\Theta(1)$: basta ritornare una variabile che contiene il numero di elementi inseriti.

Capitolo 10

Implementazione

Di seguito riporto il codice sorgente della mia implementazione dello heap binomiale.

Per più informazioni vedere [4]. In questo testo, su cui è basata la mia implementazione, ci sono alcuni diagrammi che mostrano come lavorano queste funzioni e molti suggerimenti per migliorare questa semplice implementazione.

10.1 BinomialHeap.h

52 ⟨*BinomialHeap.h* 52⟩≡

```
/*****************************************************
 *                  HEAP BINOMIALI                    *
 *  Implementazione di uno Heap Binomiale realizzata da  *
 *    ----------- Maria Luisa Guerriero -----------    *
 *****************************************************/

/* ------------------------------------------------------
* Header file contenente la dichiarazione della classe
  BinomHeap, che viene definita in BinomialHeap.cpp
* Le chiavi all'interno dello heap binomiale sono uniche.
  L'inserimento di un elemento fallisce se la chiave e` gia`
  presente.
* Lo heap binomiale e` implementato usando i templates, per cui
  puo` contenere diversi tipi di dato.
* Uno heap binomiale e` una lista di alberi binomiali. La
  rappresentazione usata per gli alberi e` con figlio sinistro-
  sibling destro.
* Questa implementazione segue la presentazione in CLR, ma con
  una operazione Union "piu` efficiente"; estrazione e
  cancellazione sono esattamente come in CLR. Il costo in una
  analisi ammortizzata e` O(1) per MakeHeap, Insert,
  Union ed IsEmpty, mentre e` O(log n) per le altre operazioni.

Le funzioni accessibili dall'esterno sono:
```

```
* MakeHeap -  Crea uno heap binomiale
* Insert(T,int) - Inserisce un elemento nello heap binomiale
* Minimum - Trova l'elemento con chiave minima
* ExtractMin - Cancella l'elemento con chiave minima
* Union - Unisce due heap binomiali dello stesso ordine
* DecreaseKey - Decrementa la chiave di un elemento
* Delete - Cancella un elemento
* Root - Ritorna un puntatore alla prima radice
* IsEmpty - Controlla se lo heap binomiale e` vuoto
* Size - Ritorna il numero di elementi dello heap binomiale

Le funzioni usate all'interno della classe sono:

* BinomHeap - Costruttore della classe BinomHeap
* Insert(BinNodo*) - Inserisce un nodo, chiamata da Insert(T,int)
* Link - Unisce due alberi binomiali con lo stesso grado
* ReverseSiblingList - Inverte la lista dei figli di un nodo
------------------------------------------------------- */

#ifndef BINOMIAL_HEAP_H
#define BINOMIAL_HEAP_H

//-------------------------------------------------------

class BinNodoBase {

public:

  int degree;
  int key;

};
//-------------------------------------------------------

template <class T> class BinNodo: public BinNodoBase {

public:

  BinNodo *parent;
  BinNodo *child;
  BinNodo *sibling;
  T data;

  BinNodo(const T& k): // Copy constructor
    data(k)
    {}
};

//-------------------------------------------------------
```

```
template <class T> class BinomHeap {

public:

  //-------------------------------------------------------------
  /* ----- FUNZIONI PUBBLICHE ----- */

  /* Crea uno heap binomiale
     OUTPUT: puntatore alla zona di memoria allocata */
  BinomHeap * MakeHeap();

  /* Inserisce un elemento nello heap binomiale
     INPUT: dato da inserire, chiave */
  void Insert(T d, int k);

  /* Trova l'elemento con chiave minima
     OUPTPUT: puntatore al nodo che lo contiene */
  BinNodo<T>* Minimum();

  /* Cancella l'elemento con chiave minima
     OUTPUT: puntatore al nodo eliminato */
  BinNodo<T>* ExtractMin();

  /* Unisce lo heap binomiale 'other' allo heap binomiale 'this'
     INPUT: puntatore allo heap binomiale da unire a questo */
  void Union(BinomHeap *other);

  /* Decrementa la chiave di un elemento
     INPUT: puntatore al nodo da modificare, nuova chiave
     OUTPUT: 1 ok, 0 errore */
  int DecreaseKey(BinNodo<T>* x, int k);

  /* Cancella un elemento
     INPUT: puntatore al nodo da cancellare */
  void Delete(BinNodo<T>* x);

  /* Ritorna un puntatore alla prima radice */
  BinNodo<T>* Root() { return head; }

  /* Controlla se lo heap binomiale e` vuoto
     OUTPUT: 1 se e` vuoto, 0 altrimenti */
  int IsEmpty() { return (head==NULL); }

  /* Ritorna il numero di elementi dello heap binomiale
     OUTPUT: num_el = numero di elementi */
  int Size() { return num_el; }

private:

  BinNodo<T> * head; //puntatore alla lista delle radici dello heap
```

```
  int num_el; // numero di elementi dello heap binomiale

  //-----------------------------------------------------------
  /* ----- FUNZIONI PRIVATE ----- */

  /* Costruttore della classe BinomHeap */
  BinomHeap();

  /* Inserisce un nodo, chiamata da Insert(T,int)
     INPUT: puntatore al nodo da inserire */
  void Insert(BinNodo<T>* x);

  /* Unisce due alberi binomiali con lo stesso grado
     INPUT: puntatori ai due alberi da collegare */
  void Link(BinNodo<T> * y, BinNodo<T> * z);

  /* Inverte la lista dei figli di un nodo
     INPUT: il nodo di cui invertire la lista dei figli
     OUTPUT: la lista dei figli in ordine inverso */
  BinNodo<T>* ReverseSiblingList(BinNodo<T>* x);

};
//-----------------------------------------------------------

#include "BinomialHeap.cpp"
#endif
```

10.2 BinomialHeap.cpp

56 ⟨*BinomialHeap.cpp* 56⟩≡

```
/* ---------------------------------------------------------
* Source file contenente la dichiarazione della classe
  BinomHeap, che viene dichiarata in BinomialHeap.h
--------------------------------------------------------- */

#ifndef BINOMIALHEAP_CPP
#define BINOMIALHEAP_CPP

#include <iostream.h>
#include <limits.h>

#include "BinomialHeap.h"

//----------------------------------------------------------
/* ----- FUNZIONI PUBBLICHE ----- */

/* Crea uno heap binomiale
   OUTPUT: puntatore alla zona di memoria allocata */

template <class T> BinomHeap<T>* BinomHeap<T>:: MakeHeap()
{
  BinomHeap *H = new BinomHeap();
  return H;
}
//----------------------------------------------------------
/* Inserisce un elemento nello heap binomiale
   INPUT: dato da inserire, chiave */
template <class T> void  BinomHeap<T>:: Insert(T d, int k)
{
  BinNodo<T>* x = new BinNodo<T>(d);
  x-> data = d;
  x-> key = k;
  Insert(x);
  num_el++;
}
//----------------------------------------------------------
/* Trova l'elemento con chiave minima
   OUPTPUT: puntatore al nodo che lo contiene */
template <class T> BinNodo<T>* BinomHeap<T>:: Minimum()
// Assume che lo heap H non sia vuoto.
// Il nodo di H con chiave minima e` una delle radici
// degli alberi binomiali in H.
{
  int min = INT_MAX;
  BinNodo<T>* ans = NULL;
  // Controlla le chiavi di tutte le radici presenti
  // nella lista degli alberi.
```

```
  for (BinNodo<T>* x = head; x != NULL; x = x->sibling)
    if (x->key < min)
    {
      min = x->key;
      ans = x;
    }
  return ans;
}
//-----------------------------------------------------------
/* Cancella l'elemento con chiave minima
   OUTPUT: puntatore al nodo eliminato */
template <class T> BinNodo<T>* BinomHeap<T>:: ExtractMin()
// Modifica H cancellando il nodo con chiave minima, mantenendo
// le proprieta` degli heap binomiali.
{
  BinNodo<T>* x = Minimum();
  BinNodo<T>* y = head;

  DecreaseKey(x, -INT_MAX);
  if (y == x)
    head = y->sibling;
  else
    while (y != NULL && y->sibling != x)
      y = y->sibling;
  y->sibling = x->sibling; // taglia fuori x
  BinomHeap *H = new BinomHeap();
  H->head = ReverseSiblingList(x->child);
  Union(H);
  num_el--;
  return x;
}
//-----------------------------------------------------------
/* Unisce due heap binomiali dello stesso ordine
   INPUT: puntatore allo heap binomiale da unire a questo */
template <class T> void BinomHeap<T>:: Union(BinomHeap<T>* other)
{
  // Fase 1. Unire due liste in un'unica lista, mantenendo le
  // proprieta` degli heap
  BinNodo<T>* x; BinNodo<T>* y; BinNodo<T>* z;
  BinNodo<T>* L1 = head;
  BinNodo<T>* L2 = other->head;

  if (L1 == NULL)
    { head = L2; return; }
  if (L2 == NULL)
    return;

  y = L1; L1 = L1->sibling;
  z = L2; L2 = L2->sibling;
```

```
  if (y->degree < z->degree)
    { x = y; y = z; }
  else
    x = z;
  head = x;
  // invariante 1: ogni volta, x e‘ l’attuale ultimo nodo nella
  // lista degli alberi dell’eventuale output
  // invariante 2: degree(x) <= degree(y) ed entrambi sono
  // <= dei gradi in L1 e L2.
  while (L1 != NULL && L2 != NULL)
  {
    // Step 1. Trova il prossimo elemento z, avanzando in una
    // delle liste
    if (L1->degree < L2->degree)
      { z = L1; L1 = L1->sibling; }
    else
      { z = L2; L2 = L2->sibling; }
    // invariante 3: degree(x) <= degree(y) <= degree (z), e
    // tutti e tre i gradi sono minori di quelli in L1, L2;
    // l’invariante 1 continua a valere.

    // Step 2. Riduzione da 3 a 2 (x, y, z ridotti ad x, y)
    if (x->degree < y->degree)
      { x->sibling = y; x = y; y = z; }
    else if (y->degree < z->degree) // && degree(x) == degree(y)
      { Link(y, x); y = z;}
    else // ( degree(y) == degree(z) && degree(x) == degree(y) )
      Link(z, y);
  }
  // Fase 2. Combinare il singolo "carry-over" albero con la
  // rimanente coda di lista.
  if (x->degree < y->degree)
    {x->sibling = y; x = y;}
  else
    Link(y, x);
  BinNodo<T>* L = (L1 == NULL) ? L2 : L1;
  // invariante 4: degree(x) <= degrees in L; l’invariante 1
  // continua a valere.
  while (L != NULL)
  {
    if (x->degree < L->degree)
      {x->sibling = L; L = NULL; } // finito
    else // possibile ulteriore "carry"
      {BinNodo<T>* w = L; L = L->sibling; Link(w, x);}
  }
  return;
}
//--------------------------------------------------------
/* Decrementa la chiave di un elemento
   INPUT: puntatore al nodo da modificare, nuova chiave
```

```
   OUTPUT: 1 ok, 0 errore */
template <class T> int BinomHeap<T>::
          DecreaseKey(BinNodo<T>* x, int k)
// Assume che x sia un nodo dello heap.
// Mette x->key = k e restaura lo heap perche‘ conservi le
// proprieta‘ degli heap.
{
  if (k > x->key)
  {
    cout << "Errore: chiave piu‘ grande della vecchia\n"<<endl;
    return 0;
  }
  x->key = k;
  BinNodo<T>* y = x;
  BinNodo<T>* z = y->parent;

  while ( z != NULL && y->key < z->key)
  // scambia le chiavi e risale lo heap
  {
    int tk = y->key; y->key = z->key; z->key = tk;
    T td = y->data; y->data = z->data; z->data = td;
    y = z; z = y->parent;
  }
  return 1;
}
//-----------------------------------------------------------
/* Cancella un elemento
   INPUT: puntatore al nodo da cancellare */
template <class T> void BinomHeap<T>:: Delete(BinNodo<T>* x)
{
  DecreaseKey(x, -INT_MAX);
  ExtractMin();
}
//-----------------------------------------------------------
/* ----- FUNZIONI PRIVATE ----- */

/* Costruttore della classe BinomHeap */
template <class T> BinomHeap<T>:: BinomHeap()
{
  head = NULL;
  num_el = 0;
}
//-----------------------------------------------------------
/* Inserisce un nodo, chiamata da Insert(T, int)
   INPUT: puntatore al nodo da inserire */
template <class T> void BinomHeap<T>:: Insert(BinNodo<T>* nodo)
{
  BinomHeap *H = new BinomHeap();
  nodo->parent = NULL;
  nodo->child = NULL;
```

```
  nodo->sibling = NULL;
  nodo->degree = 0;
  H->head = nodo;
  Union(H);
}
//-----------------------------------------------------------
/* Unisce due alberi binomiali con lo stesso grado
   INPUT: puntatori ai due alberi da collegare */
template <class T> void BinomHeap<T>::
          Link(BinNodo<T>* y, BinNodo<T>* z)
// Assume che y e z siano alberi binomiali dello stesso grado k.
// Li unisce in un albero binomiale di grado  k+1 radicato in z.
// Quindi, l'albero risultante rimpiazza z nella lista degli
// alberi in cui era, se esiste.
{
  if (y->key < z->key)
  {
    int tk = z->key; z->key = y->key; y->key = tk;
    T td = z->data; z->data = y->data; y->data = td;
    BinNodo<T>* tc = z->child; z->child = y->child; y->child = tc;
  }
  y->parent = z; y->sibling = z->child; z->child = y; z->degree++ ;
  // Nota: sibling(z) non cambia, quindi non e` affetta
  // l'appartenenza di z ad una lista di alberi.
}
//-----------------------------------------------------------
/* Inverte la lista dei figli di un nodo
   INPUT: il nodo di cui invertire la lista dei figli
   OUTPUT: la lista dei figli in ordine inverso */
template <class T> BinNodo<T>* BinomHeap<T>::
          ReverseSiblingList(BinNodo<T>* x)
{
  BinNodo<T>* r = NULL; // la parte invertita della lista
  while (x != NULL)
  {
    BinNodo<T>* y = x; x = x->sibling;
    y->sibling = r; r = y;
  }
  return r;
}

#endif
//-----------------------------------------------------------
```

Ho inserito anche due file che possono essere utili per capire come va usato lo heap binomiale.

10.3 TestHeap.cpp

85 ⟨*TestHeap.cpp* 85⟩≡

```
/*********************************************************************
 *                          HEAP BINOMIALI                           *
 *File di test e di esempio d'uso dello Heap Binomiale realizzato da*
 *         ----------- Maria Luisa Guerriero -----------             *
 *********************************************************************/
/* ------------------------------------------------------------
* In questo esempio lo heap binomiale e` utilizzato per contenere
  delle stringhe (string), ma si potrebbe usare anche per
  inserire altri tipi di dato (ad esempio int, float,...).
------------------------------------------------------------- */

#include <iostream.h>
#include <string>

#include "BinomialHeap.h"

//------------------------------------------------------------
void main()
{
 cout<<"Creo uno heap binomiale per contenere stringhe(string)\n";
 BinomHeap<string> *H = H->MakeHeap();
 string str;
 int key;

 cout << "\nInserisco un elemento (\"prova\") ";
 cout << "con la sua chiave (10)...\n";
 str="prova";
 key=10;
 H->Insert(str, key);

 cout << "...ne inserisco un altro (\"heap\") ";
 cout << "con la sua chiave (3)...\n";
 str="heap";
 key=3;
 H->Insert(str, key);

 cout << "...ed infine l'ultimo (\"binomiali\") ";
 cout << " con la sua chiave(5).\n";
 str="binomiali";
 key=5;
 H->Insert(str, key);

 cout << "Stampo le informazioni sullo heap binomiale\n";
```

```
 if (!H->IsEmpty())
 {
   cout << "minimo = " << H->Minimum()->data;
   cout << "  chiave = " << H->Minimum()->key << endl;

   cout << "radice = " << H->Root()->data;
   cout << "  chiave = " << H->Root()->key << endl;

   cout << "size = " << H->Size() << endl;
 }
 cout << "\nCancello l'elemento con chiave minima...\n";
 if (!H->IsEmpty())
   H->ExtractMin(); // equivale a H->Delete(H->Minimum());
 cout << "...e stampo nuovamente le informazioni sullo heap.\n";
 if (!H->IsEmpty())
 {
   cout << "minimo = " << H->Minimum()->data;
   cout << "  chiave = " << H->Minimum()->key << endl;

   cout << "radice = " << H->Root()->data;
   cout << "  chiave = " << H->Root()->key << endl;

   cout << "size = " << H->Size() << endl;
 }
 else cout << "heap vuoto";

 if (!H->IsEmpty())
 {
   cout << "\nOra decremento la chiave del minimo portandola a 2\n";
   key=2;
   int i = H->DecreaseKey(H->Minimum(),key);
   if (i!=0)
   {
     cout << "minimo = " << H->Minimum()->data;
     cout << "  nuova chiave = " << H->Minimum()->key << endl;
   }
 }
 cout << "\nCancello gli altri elementi...\n";
 if (!H->IsEmpty()) H->Delete(H->Minimum());
 if (!H->IsEmpty()) H->Delete(H->Minimum());

 cout << "...e controllo che lo heap sia vuoto.\n";
 if (!H->IsEmpty())
 {
   cout << "radice = " << H->Root()->data;
   cout << "  chiave = " << H->Root()->key << endl;
 }
 else cout << "heap vuoto, size = " << H->Size() <<endl;
}
//-----------------------------------------------------------
```

10.4 TestInterattivoHeap.cpp

87 ⟨*TestInterattivoHeap.cpp* 87⟩≡

```
/*****************************************************************
 *                         HEAP BINOMIALI                        *
 *File per un utilizzo guidato dello Heap Binomiale realizzato da*
 *         ----------- Maria Luisa Guerriero -----------         *
 *****************************************************************/
/* --------------------------------------------------------
* In questo esempio lo heap binomiale e‘ utilizzato per contenere
  delle stringhe (string), ma si potrebbe usare anche per
  inserire altri tipi di dato (ad esempio int, float,...).
-------------------------------------------------------- */
#include <iostream.h>
#include <string>

#include "BinomialHeap.h"
#define STRLEN 80
//---------------------------------------------------------
int main(void)
{
  BinomHeap<string> *H = H->MakeHeap();
  string str;
  int key, i=0;
  char sc;

  cout <<"In questo esempio nelllo heap sono memorizzate ";
  cout <<"stringhe (string), ma si possono inserire altri ";
  cout <<"tipi di dati (ad es. int, float,...)\n";

  while(1)
  {
    cout << "1) Inserire un elemento\n";
    cout << "2) Size\n";
    cout << "3) Radice\n";
    cout << "4) Cancellare minimo\n";
    cout << "5) Minimo\n";
    cout << "6) Decrease key (minimo)\n";
    cout << "7) Uscire\n";

    cin >> sc;
    switch (sc)
    {
      /* 1) Inserire un elemento */
      case '1': cin >>str;
                cin >>key;
                H->Insert(str, key);
                break;

      /* 1) Size */
```

```
      case '2': if (!H->IsEmpty())
                  cout << "size = " << H->Size() << endl;
                else cout << "heap vuoto" << endl;
                break;

      /* 3) Radice */
      case '3': if (!H->IsEmpty())
                {
                  cout << "radice = " << H->Root()->data;
                  cout << "  chiave = " << H->Root()->key << endl;
                }
                else cout << "heap vuoto" << endl;
                break;

      /* 4) Cancellare minimo */
      case '4': if (!H->IsEmpty()) H->ExtractMin();
                       //equivale a H->Delete(H->Minimum());
                else cout << "heap vuoto" << endl;
                break;

      /* 5) Minimo */
      case '5': if (!H->IsEmpty())
                {
                  cout << "minimo = " << H->Minimum()->data;
                  cout << "  chiave = " << H->Minimum()->key << endl;
                }
                else cout << "heap vuoto" << endl;
                break;

      /* 6) Decrease key (minimo) */
      case '6': if (!H->IsEmpty())
                {
                  cin >>key;
                  int i = H->DecreaseKey(H->Minimum(),key);
                  if (i!=0)
                  {
                    cout << "minimo = " << H->Minimum()->data;
                    cout << "  nuova chiave = ";
                    cout << H->Minimum()->key << endl;
                  }
                 }
                else cout << "heap vuoto" << endl;
                break;

      /* 7) Uscire */
      case '7': exit(0);
    }
  }
}
//-----------------------------------------------------------
```

Parte III

B-ALBERI BINARI

Capitolo 11

Introduzione

Abbiamo visto come le tabelle hash si prestino bene a realizzare dizionari perché permettono di eseguire molto velocemente le operazioni *Find*, *Insert* e *Delete*; gli heap invece permettono di realizzare velocemente le operazioni delle code con priorità come *Minimum* ed *ExtractMin*.

Gli **alberi bilanciati di ricerca** permettono di realizzare tutte le operazioni citate sopra in tempo $\mathrm{O}(\log N)$ nel caso pessimo. I B-alberi binari sono un tipo particolare di alberi bilanciati di ricerca.

Ma a cosa possono servire queste strutture? Negli anni sono state proposte molte tecniche per organizzare i file in modo da aumentare la velocità di ricerca al loro interno (associando ad ogni file un file indice, il quale, se troppo grande, può essere a sua volta indicizzato, e così via); in [8] Knuth esamina le principali tecniche. Sebbene non ci sia una tecnica che sia ottima per ogni applicazione, quella basata sulla struttura detta **B-albero** (B-tree), o su una delle sue tante varianti, come ad esempio i B-alberi binari, è ampiamente usata. Il B-albero, di fatto, è l'organizzazione standard per gli indici in un sistema di database.

Il punto di partenza della mia analisi è una struttura dati usata per la memoria interna chiamata albero binario di ricerca. Poi vedremo i ben più interessanti alberi bilanciati di ricerca: prima i B-alberi, poi gli alberi 2-3 (che non sono altro che dei B-alberi di ordine 1); quindi tratteremo in dettaglio i B-alberi binari. Infine, come sempre, c'è la mia implementazione in C++ di questi ultimi.

Capitolo 12

Gli alberi di ricerca

12.1 Alberi binari di ricerca

Un albero binario di ricerca è un albero in cui:

- per ogni nodo u, tutti gli elementi contenuti nel sottoalbero radicato nel figlio sinistro di u sono minori dell'elemento contenuto in u
- per ogni nodo u, tutti gli elementi contenuti nel sottoalbero radicato nel figlio destro di u sono maggiori dell'elemento contenuto in u

In questo modo, per effettuare una ricerca basta confrontare l'elemento da cercare con quello contenuto nella radice: se questi sono diversi, il procedimento viene riapplicato al sottoalbero sinistro se la chiave cercata è minore, o a quello destro se è maggiore.

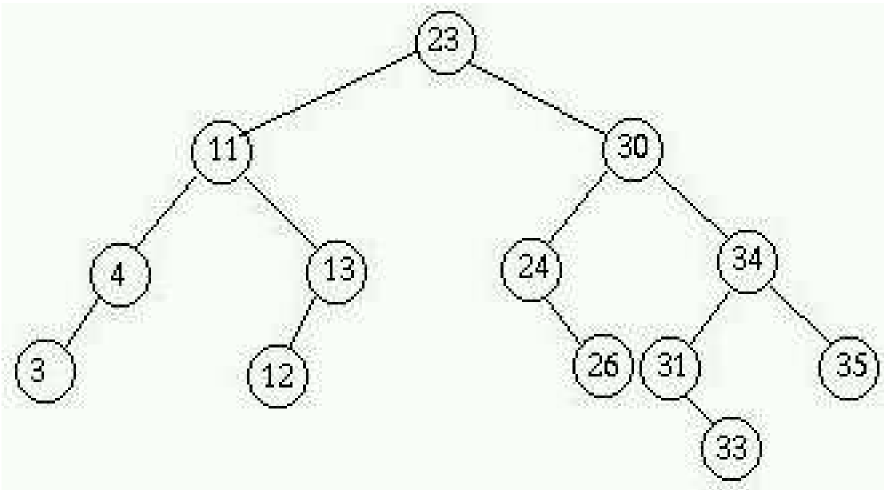

Figura 12.1: Albero binario di ricerca

Usando un albero binario di ricerca *Find*, *Insert*, *Delete*, *Minimum*, *ExtractMin*, *Maximum* e *ExtractMax* hanno tutte complessità $O(L)$, dove L è il livello massimo delle foglie. Nel caso pessimo, però, L è $O(N)$, perché i cammini radice-foglia non sono necessariamente tutti della stessa lunghezza. Per questo motivo sono state trovate delle strutture **"bilanciate"** in cui L è $O(\log N)$, come i B-alberi.

12.2 B-alberi

I B-alberi furono proposti negli anni '70 da R. Bayer e E. McCreight in [1].

I nodi di un B-albero possono avere un numero di chiavi diverso da nodo a nodo, ma con dei limiti inferiori e superiori. Premesso che esistono diverse definizioni di B-albero, che si differenziano principalmente proprio per come vengono calcolati questi limiti, ecco quella più frequente: ogni nodo di un B-albero di **ordine** m contiene al massimo $2m$ chiavi e $2m + 1$ figli, ed almeno m chiavi e $m + 1$ figli. La conseguenza di questa proprietà è che ogni nodo è almeno pieno a metà.

Come già accennato, l'utilizzo che viene fatto dei B-alberi è quello di memorizzare dizionari o indici di file in memoria secondaria (che ha un tempo di accesso molto elevato rispetto alla memoria principale). Una scelta opportuna di m permette di memorizzare un nodo in una pagina, che può essere trasferita interamente dalla memoria secondaria a quella principale con un solo accesso, e di effettuare poi nella memoria principale la ricerca binaria (la cui complessità è $O(\log m)$) all'interno del nodo. Di solito infatti nodi grandi, contenenti molte chiavi, non possono essere tenuti in memoria principale e richiedono un accesso alla memoria secondaria ogni volta che sono analizzati; comunque il fatto di mantenere più di una chiave per ogni nodo diminuisce il costo delle operazioni di ricerca, inserimento e cancellazione.

Nella usuale rappresentazione un nodo contiene un record del file indice, ha una lunghezza fissa che può contenere $2m$ chiavi e $2m + 1$ puntatori ai figli, e contiene informazioni aggiuntive sul numero di chiavi memorizzate nel nodo.

La bontà dei B-alberi sta nel fatto che l'inserimento e la cancellazione di elementi lascia il B-albero bilanciato, cioè con tutte le foglie allo stesso livello; ad esempio l'inserimento di nuovi record in un file ad accesso casuale lascia il B-albero bilanciato, a differenza di quello che succederebbe usando un semplice albero di ricerca.

Il livello massimo delle foglie in un B-albero in cui sono memorizzate N chiavi è $O(\log_d N)$, dove d è l'ordine del B-albero, e perciò il cammino più lungo al suo interno contiene al massimo $O(\log_d N)$ nodi.

Una ricerca in un file indicizzato da un albero non bilanciato di N record potrebbe visitare N nodi, mentre usando un B-albero non visita mai più di $1 + \log_d N$ nodi. Poiché ogni visita richiede un accesso alla memoria secondaria, il bilanciamento dell'albero riduce il tempo di ricerca.

Sono stati proposti molti schemi per bilanciare gli alberi (vedere ad esempio [6], [7] e [12]). Ovviamente è necessario del tempo per effettuare il bilanciamento, perciò bisogna fare in modo che il risparmio di tempo nella ricerca sia almeno maggiore del costo dello stesso bilanciamento.

Si può dimostrare che il costo di una ricerca è $O(log_d \frac{n+1}{2})$. Anche inserimento e cancellazione impiegano $O(\log_d N)$ tempo nel caso pessimo.

Grazie al fatto che cresce come il logaritmo della dimensione del file, il costo di una operazione di ricerca resta contenuto anche per file molto grandi, come mostra la tabella seguente. Ad esempio supponendo di avere un B-albero di ordine 50 che indicizza un file di un milione di record (10^9), bastano nel caso pessimo solo $\log_{50} 1.000.000 \cong 4$ accessi alla memoria secondaria per ricercare un elemento. In realtà esistono tecniche di implementazione che riducono a 3 il numero di accessi nel caso pessimo, e ad ancora meno quelli nel caso medio.

-	10^3	10^4	10^5	10^6	10^7
10	3	4	5	6	7
50	2	3	3	4	4
100	2	2	3	3	4
150	2	2	3	3	4

Tabella 12.1: Numero di accessi nel caso pessimo in base alle dimensioni dei nodi e del file — Colonne: dimensione file (numero di record) — Righe: dimensione nodi

A questo punto il vantaggio di avere nodi che contengono un grande numero di chiavi dovrebbe essere chiaro: come aumenta in fattore di branch d, la base del logaritmo aumenta, e perciò il costo delle operazioni di ricerca, inserimento e cancellazione diminuisce.

Ci sono però limiti pratici alla dimensione dei nodi: la maggior parte dei sistemi hardware limita la quantità di dati che può essere trasferita con un accesso alla memoria secondaria. Inoltre ogni dispositivo ha una dimensione fissa delle tracce che deve essere rispettata per non sprecare grandi quantità di spazio. Perciò in pratica la dimensione ottima dei nodi dipende notevolmente dalle caratteristiche del sistema e dei dispositivi su cui è salvato il file.

Per quanto riguarda l'uso della memoria, abbiamo già detto che i B-alberi garantiscono il 50% di utilizzo della memoria, allocando e rilasciando spazio quando il file cresce o diminuisce di dimensione. Inoltre non è mai necessaria una massiccia "riorganizzazione" del file, come invece succede per esempio usando le tabelle hash.

Fino ad ora abbiamo considerato l'**accesso casuale** al file, che avviene specificando una chiave. Spesso però si vuole vedere il file in modo **sequenziale**, usando una operazione (chiamiamola "next") che permetta di scandire tutti i record in sequenza in base all'ordinamento tra chiavi.

Sfortunatamente però un B-albero non funziona bene con l'accesso sequenziale. Una visita anticipata, ad esempio, estrae tutte le chiavi in ordine, ma richiede spazio per almeno $h = \log_i(n + 1)$ nodi nella memoria principale dato che mette sullo stack i nodi lungo il cammino dalla radice per evitare di leggerli due volte. Inoltre una operazione next può dover attraversare diversi nodi prima di raggiungere la chiave desiderata. Per esempio, la chiave più piccola è memorizzata nella foglia più a sinistra e per trovarla bisogna accadere a tutti i nodi che sono nel cammino tra la radice e questa foglia. Una variazione dei B-alberi, i **B+-alberi**, consentono un efficiente accesso sequenziale, mantenendo il costo logaritmico delle operazioni di ricerca, inserimento e cancellazione.

12.3 Alberi 2-3

Un **albero 2-3** è un B-albero di ordine 1: ogni nodo può quindi avere 2 o 3 figli e contenere 1 o 2 chiavi. Questa struttura fu introdotta da Hopcroft ed è molto utile nella memorizzazione ad un livello. La piccola dimensione dei nodi rende gli alberi 2-3 non pratici per la memoria esterna, ma abbastanza adatti per una struttura dati interna.

Formalizzando, le proprietà che caratterizzano un albero 2-3 sono:

- ogni nodo interno (cioè non foglia) ha k figli, con k uguale a 2 o 3, e contiene $k-1$ chiavi, ordinate in senso crescente
- le foglie sono tutte allo stesso livello e non contengono alcuna chiave
- se un nodo u ha k figli, allora l'i-esima chiave di u è maggiore di tutte le chiavi contenute nel sottoalbero la cui radice è l'i-esimo figlio di u, ed è minore di tutte le chiavi contenute nel sottoalbero la cui radice è l'$(i+1)$-esimo figlio di u, per $1 \leq i \leq k-1$

Vediamo di chiarire eventuali dubbi con l'esempio in Figura 12.2:

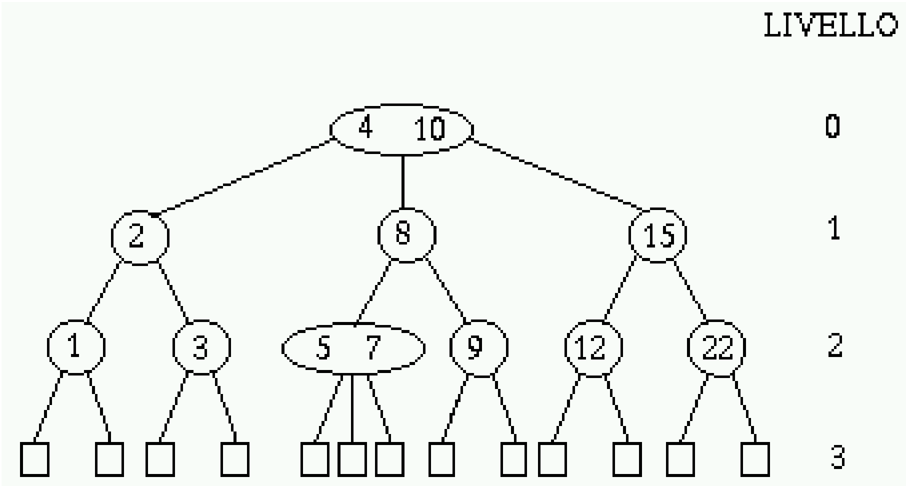

Figura 12.2: Un albero 2-3

Grazie a queste proprietà il livello delle foglie L non è più $O(N)$ nel caso pessimo, ma:

$$\lceil \log_3(n+1) \rceil \leq L \leq \lceil \log_2(n+1) \rceil$$

Il sistema che si usa nella ricerca è concettualmente lo stesso degli alberi binari di ricerca: si confronta l'elemento da cercare con quelli contenuti nella radice e, se l'elemento non è presente, ma è maggiore dell'i-esimo, ma minore dell'$(i+1)$-esimo elemento della radice, riapplicare il procedimento all'$(i+1)$-esimo sottoalbero fino a che si trova l'elemento (successo) o si arriva ad una foglia (fallimento).

Vediamo un esempio: volendo ricercare nell'albero di Figura 12.2 la chiave 3, dalla radice sceglieremmo il cammino di sinistra (perché $3 < 4$). Per cercare una chiave k tale che $4 < k < 10$ verrebbe scelto il cammino centrale, mentre seguiremmo quello di destra per chiavi $k > 10$, e così via per ogni nodo incontrato.

La complessità delle operazioni *Find*, *Insert*, *Delete*, *Minimum*, *ExtractMin*, *Maximum* e *ExtractMax* è $O(L) = O(\log N)$.

Capitolo 13

B-alberi binari

Un'altra variante dei B-alberi è rappresentata dai **B-alberi binari**, i quali, come gli alberi 2-3, sono adatti alla memorizzazione ad un livello, ma hanno in più il vantaggio della semplicità con cui si possono implementare.

Infatti i B-alberi non sono affatto banali da implementare; tutte le operazioni degli alberi 2-3, invece, sono semplici, con la sola eccezione della *Delete*. È infatti proprio per semplificare questa operazione che sono stati introdotti i B-alberi binari, che sono essenzialmente B-alberi di ordine 1, e che permettono di rappresentare un albero 2-3 con un albero binario di ricerca in cui il livello delle foglie resta $\Theta(\log N)$.

Ogni nodo di un B-albero binario ha 1 o 2 chiavi e 2 o 3 puntatori a figli. Per evitare lo spreco di spazio di nodi solo pieni a metà, si usa una rappresentazione collegata. Un B-albero binario è un albero binario di ricerca T ottenuto trasformando un albero 2-3 D: un nodo di D, detto **"supernodo"**, è rappresentato con 1 o 2 nodi binari di T. Una connessione padre-figlio tra due nodi binari è detta orizzontale se è interna allo stesso supernodo, o verticale se è tra due supernodi. Solo una connessione tra un nodo ed il suo figlio destro (detta connessione destra) può essere orizzontale. In altre parole, un supernodo di D con 2 figli è rappresentato in T con un nodo normale con un figlio destro ed un figlio sinistro; un supernodo di D con 3 figli, invece, è rappresentato in T con 2 nodi binari di cui uno è la radice del supernodo ed ha come figlio destro (tramite una connessione orizzontale destra) l'altro.

Ad ogni nodo u dell'albero T è associato il livello del supernodo dell'albero 2-3 che u rappresenta; conviene numerare i livelli a partire dalle foglie, in modo che queste siano al livello 0, e la radice sia al livello L. Notare che due nodi con una connessione orizzontale destra (detti **"sibling"**) sono allo stesso livello, poiché rappresentano lo stesso supernodo di D.

Poiché L è $\Theta(\log N)$, un qualsiasi percorso radice-foglia comprende al massimo $2L$ nodi interni più la foglia, ed è quindi lungo $\Theta(\log N)$.

Perciò inserzione, cancellazione e ricerca compiono $O(\log_2 N)$ passi così come nei B-alberi; notare che dato che le connessioni orizzontali possono essere solo destre, una ricerca effettuata nel cammino più a destra può toccare il doppio di nodi del cammino più a sinistra.

Le proprietà che caratterizzano un B-albero binario e che devono sempre essere rispettate sono:

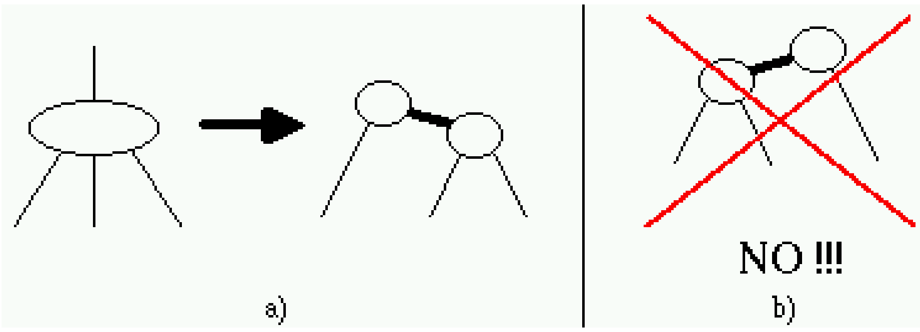

Figura 13.1: a) Rappresentazione di un supernodo di un albero 2-3 con 2 nodi con connessione orizzontale destra. b) Le connessioni orizzontali sinistre sono vietate

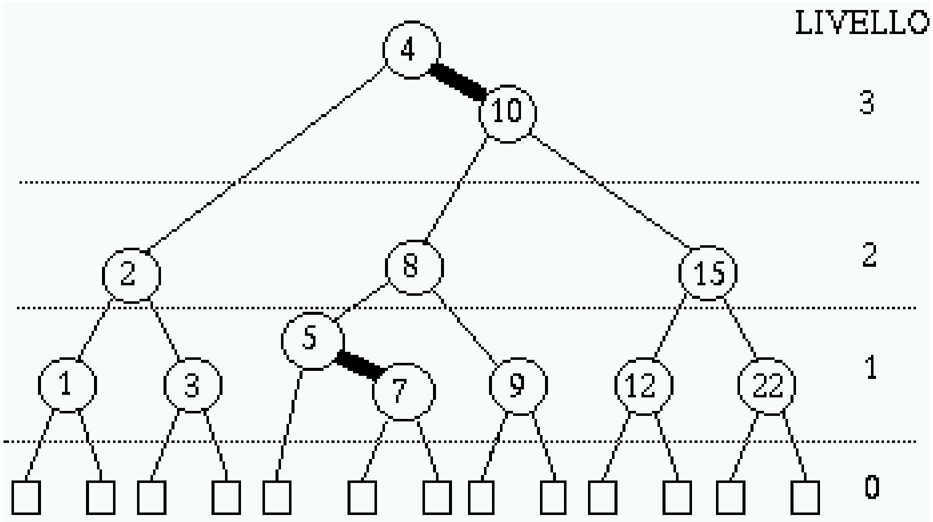

Figura 13.2: Il B-albero binario che rappresenta l'albero 2-3 di Figura 12.2

- una connessione orizzontale può essere solo destra
- un supernodo non può essere formato da più di due nodi binari con connessioni orizzontali
- in una connessione verticale livello(padre)=livello(figlio) + 1

Una estensione dei B-alberi binari, che permette sia al link destro che a quello sinistro di puntare ad un nodo sibling, aggiunge la simmetria che non è presente nei B-alberi binari. Perciò sono stati chiamati **B-alberi binari simmetrici**.

13.1 Operazioni sui B-alberi binari

Con questa struttura dati le già citate operazioni *Find*, *Insert*, *Delete*, *Minimum*, *ExtractMin*, *Maximum* ed *ExtractMax* hanno tutte complessità $O(\log N)$ e sono abbastanza facili da implementare: l'unica difficoltà consiste nell'assicurare che non ci siano mai due o più link a sibling successivi. Vediamole una per volta:

- **BBTree** - Costruttore: crea un B-albero binario:
  O(1) → alloca un puntatore ad un nodo ed inizializza le variabili globali
- **IsEmpty** - Controlla se il B-albero binario è vuoto:
  O(1) → controlla se il puntatore alla radice è uguale a NULL
- **Height** - Ritorna l'altezza dell'albero:
  O(1) → L'altezza albero è il livello della radice
- **Size** - Ritorna la dimensione (il numero di elementi) dell'albero:
  O(L) → Visita simmetrica dell'albero
- **Insert** - Inserisce un elemento nel B-albero binario:
  O(L) → Inserisce in fondo. Richiama *RightRotate* e *LeftRotate* per ogni nodo incontrato nella risalita. Se l'albero è vuoto, invece, inserisce il nodo direttamente come radice
- **Delete** - Cancella un elemento dal B-albero binario:
  O(L) → Discende lungo l'albero, cancella l'elemento e ribilancia durante la risalita eseguendo 3 *RightRotate* e 2 *LeftRotate* per ogni nodo incontrato
- **Find** - Cerca un elemento nel B-albero binario:
  O(L) → Visita simmetrica dell'albero
- **Root** - Ritorna la chiave della radice:
  O(1) → Legge la variabile contenente la chiave
- **Minimum** - Trova l'elemento con chiave minima:
  O(L) → Scende sempre a sinistra fino in fondo all'albero
- **Maximum** - Trova l'elemento con chiave massima:
  O(L) → Scende sempre a destra fino in fondo all'albero
- **Show** - Visualizza gli elementi contenuti nell'albero:
  O(L) → Visita simmetrica dell'albero
- **LeftChild** - Ritorna un puntatore al figlio sx del nodo:
  O(1) → Legge la variabile contenente il puntatore al figlio sinistro
- **RightChild** - Ritorna un puntatore al figlio dx del nodo:
  O(1) → Legge la variabile contenente il puntatore al figlio destro

dove L è il livello massimo delle foglie = O(log N)

Capitolo 14

Implementazione in C++

La mia implementazione è basata su quella esposta da Bertossi in [2].

14.1 BBTree.h

75 $\langle$*BBTree.h* 75$\rangle\equiv$

```
/*****************************************************
 *                  B-ALBERI BINARI                  *
 * Implementazione di un B-albero binario realizzata da *
 *    ----------- Maria Luisa Guerriero -----------    *
 *****************************************************/

/* -----------------------------------------------------
* Header file contenente la dichiarazione delle classi nodo
  e BBTree, che viene definita in BBTree.cpp
* Le chiavi all'interno del B-albero binario sono uniche.
  L'inserimento di un elemento fallisce se la chiave e` gia`
  presente.
* Il B-albero binario e` implementato usando i templates,
  per cui puo` contenere diversi tipi di dato.

Le funzioni accessibili dall'esterno sono:

* BBTree - Costruttore: crea un B-albero binario
* IsEmpty - Controlla se il B-albero binario e` vuoto
* Height() - Ritorna il livello della radice dell'albero
* Size() - Ritorna il numero di elementi presenti nell'albero
* Insert(T) - inserisce un elemento nel B-albero binario
* Delete(T) - Cancella un elemento dal B-albero binario
* Find(T) - Cerca un elemento nel B-albero binario
* Root - Ritorna la chiave della radice
* Minimum() - Trova l'elemento con chiave minima
* Maximum() - Trova l'elemento con chiave massima
* Show() - Visualizza gli elementi contenuti nell'albero
* LeftChild - Ritorna un puntatore al figlio sx del nodo
```

```
* RightChild - Ritorna un puntatore al figlio dx del nodo

Le funzioni usate all'interno della classe sono:

* Height(nodo*) - Ritorna il livello del nodo che le viene
  passato (la chiama Height())
* Size(nodo*) - Calcola il numero di elementi del sottoalbero
  radicato nel nodo che le viene passato (la chiama Size())
* Insert(T,nodo*&) - Inserisce un elemento (viene chiamata
  da Insert(T) se l'albero non e` vuoto)
* InsertRoot - Inserisce un elemento come radice (chiamata
  da Insert(T) se l'albero e` vuoto)
* Delete(T,nodo*&) - Cancella un elemento (chiamata da Delete(T))
* Read - Legge la chiave del nodo che le viene passato
* Find(T,nodo*) - Cerca un elemento nel B-albero binario (viene
  chiamata da Find(T))
* Minimum(nodo*) - Trova l'elemento con chiave min del sottoalbero
  radicato nel nodo che le viene passato (la chiama Minimum())
* Maximum(nodo*) - Trova l'elemento con chiave max del sottoalbero
  radicato nel nodo che le viene passato (la chiama Maximum())
* Show(nodo*) - Visualizza gli elementi del sottoalbero radicato
  nel nodo passato (la chiama Show())
* RightRotate - Esegue una rotazione destra
* LeftRotate - Esegue una rotazione sinistra
----------------------------------------------------------- */

#ifndef BBTREE_H
#define BBTREE_H

//-----------------------------------------------------------
/* Classe base per rappresentare un nodo del B-albero binario */
class nodoBase {

public:
  int level; //livello del nodo

};
//-----------------------------------------------------------
/* Classe per rappresentare un nodo del B-albero binario */
template <class T> class nodo: public nodoBase {

public:

  nodo<T> * left; // puntatore al figlio sinistro
  nodo<T> * right; // puntatore al figlio destro
  T key; // chiave dell'elemento contenuto nel nodo

  nodo(const T& k): // Copy constructor
    key(k)
    {}
```

```
};
//---------------------------------------------------------
/* ---- Classe per rappresentare il B-albero binario ---- */
template <class T> class BBTree {

public:

  //---------------------------------------------------------
  /* ----- FUNZIONI PUBBLICHE ----- */

  /* Costruttore: crea un B-albero binario */
  BBTree();

  /* Controlla se il B-albero binario e` vuoto
     OUTPUT: 1 se e` vuoto, 0 altrimenti */
  int IsEmpty() { return (BT) ? 0 : 1; }

  /* Ritorna l'altezza dell'albero
     OUTPUT: livello della radice = altezza dell'albero */
  int Height() { if(!IsEmpty()) return Height(BT); }

  /* Ritorna la dimensione (numero di elementi) dell'albero
     OUTPUT: size = dimensione dell'albero */
  int Size();

  /* Inserisce un elemento nel B-albero binario
     INPUT: chiave dell'elemento da inserire */
  void Insert(T k);

  /* Cancella un elemento dal B-albero binario
     INPUT: chiave dell'elemento da cancellare */
  void Delete(T k) { Delete(k, BT); if (BT==bottom) BT=NULL; }

  /* Cerca un elemento nel B-albero binario
     INPUT: chiave dell'elemento da cercare
     OUTPUT: 1 se trova l'elemento, 0 altrimenti */
  int Find(T k) { return Find(k, BT); }

  /* Legge la chiave della radice
     OUTPUT: chiave letta */
  T Root() { return BT->key; }

  /* Trova l'elemento con chiave minima
     OUTPUT: chiave letta */
  T Minimum() { return Minimum(BT); }

  /* Trova l'elemento con chiave massima
     OUTPUT: chiave letta */
  T Maximum() { return Maximum(BT); }
```

```
  /* Visualizza gli elementi contenuti nell'albero */
  void Show() { Show(BT); }

  /* Ritorna un puntatore al figlio sx del nodo passato
     INPUT: nodo
     OUTPUT: figlio sx del nodo */
  nodoBase* LeftChild(nodoBase* p) { return p->left; }

  /* Ritorna un puntatore al figlio dx del nodo passato
     INPUT: nodo
     OUTPUT: figlio dx del nodo */
  nodoBase* RightChild(nodoBase* p) { return p->right; }

private:

  nodo<T> * BT; // puntatore alla radice del B-albero binario
  nodo<T> * bottom; // sentinella (rappresenta le foglie)
  nodo<T> * deleted; // usata nella cancellazione di un nodo
  nodo<T> * last;    //                  "
  int size; //dimensione del B-albero binario

  //--------------------------------------------------------
  /* ----- FUNZIONI PRIVATE ----- */

  /* Ritorna il livello del nodo passato (la chiama Height())
     INPUT: nodo
     OUTPUT: livello del nodo */
  int Height (nodo<T>* p) { return p->level; }

  /* Calcola il numero di elementi del sottoalbero radicato nel
     nodo che le viene passato (la chiama Size())
     INPUT: nodo  */
  void Size (nodo<T>* p);

  /* Inserisce un elemento (viene chiamata da Insert(T)
     se l'albero non e` vuoto)
     INPUT: chiave dell'elemento,sottoalbero in cui inserire */
  void Insert(T k, nodo<T>*& btree);

  /* Inserisce un elemento come radice(chiamata da Insert(T)
     se l'albero e` vuoto)
     INPUT: chiave dell'elemento da inserire */
  void InsertRoot(T k);

  /* Cancella un elemento (viene chiamata da Delete(T))
     INPUT: chiave dell'elemento,sottoalbero da cui eliminare */
  void Delete(T k, nodo<T>*& btree);

  /* Legge la chiave del nodo che le viene passato
```

```
     INPUT: nodo
     OUTPUT: chiave letta */
  T Read(nodoBase * btree) { if (btree !=NULL) return btree->key; }

  /* Cerca un elemento nel B-albero binario (viene chiamata da
     Find(T))
     INPUT: chiave dell'elemento,sottoalbero in cui cercare
     OUTPUT: 1 se trova l'elemento, 0 altrimenti */
  int Find(T k, nodo<T>* p);

  /* Trova l'elemento con chiave min del sottoalbero radicato
     nel nodo che le viene passato (la chiama Minimum())
     INPUT: nodo
     OUTPUT: chiave dell'elemento minimo */
  T Minimum(nodo<T>* p);

  /* Trova l'elemento con chiave max del sottoalbero radicato
     nel nodo che le viene passato (la chiama Maximum())
     INPUT: nodo
     OUTPUT: chiave dell'elemento massimo */
  T Maximum(nodo<T>* p);

  /* Visualizza gli elementi del sottoalbero radicato nel nodo
     che le viene passato (la chiama Show())
     INPUT: nodo */
  void Show(nodo<T> * btree);

  /* Esegue una rotazione destra
     INPUT: nodo */
  void RightRotate(nodo<T>*& btree);

  /* Esegue una rotazione sinistra
     INPUT: nodo */
  void LeftRotate(nodo<T>*& btree);

};
//-----------------------------------------------------------

#include "BBTree.cpp"
#endif
```

14.2 BBTree.cpp

80 ⟨*BBTree.cpp* 80⟩≡

```
/* -----------------------------------------------------------
* Source file contenente la definizione della classe BBTree,
  che viene dichiarata in BBTree.h
----------------------------------------------------------- */

#ifndef BBTREE_CPP
#define BBTREE_CPP

#include "BBTree.h"

//-----------------------------------------------------------
/* ----- FUNZIONI PUBBLICHE ----- */

/* Costruttore: crea un B-albero binario */
template <class T> BBTree<T>:: BBTree()
{
  bottom = (nodo<T>*) new nodoBase;
  bottom->level = 0;
  bottom->left = bottom;
  bottom->right = bottom;
  deleted = bottom;
  BT = NULL;
}
//-----------------------------------------------------------
/* Ritorna la dimensione (numero di elementi) dell'albero
   OUTPUT: size = dimensione dell'albero */
template <class T> int BBTree<T>::Size ()
{
  size = 0;
  if (!IsEmpty())
    Size(BT);
  return size;
}
//-----------------------------------------------------------
/* Inserisce un elemento nel B-albero binario
   INPUT: chiave del nodo da inserire */
template <class T> void BBTree<T>:: Insert(T k)
{
  if (!IsEmpty())
    Insert(k,BT);
  else InsertRoot(k);
}
//-----------------------------------------------------------
/* ----- FUNZIONI PRIVATE ----- */

/* Calcola il numero di elementi del sottoalbero radicato nel
   nodo che le viene passato (la chiama Size())
```

```
   INPUT: nodo */
template <class T> void BBTree<T>::Size (nodo<T>* p)
{
  if (p != bottom)
  {
    if (p->left != bottom)
      Size (p->left);
    size +=1;
    if (p->right != bottom)
      Size (p->right);
  }
}
//-----------------------------------------------------------
/* Inserisce un elemento (viene chiamata da Insert(T)
   se l'albero non e` vuoto)
   INPUT: chiave dell'elemento,sottoalbero in cui inserire */
template <class T> void BBTree<T>:: Insert(T k, nodo<T>*& btree)
{
  if (btree==bottom)
  {
    btree = new nodo<T>(k);
    btree->key = k;
    btree->left = bottom;
    btree->right = bottom;
    btree->level = 1;
  }
  else
  {
    if (k< btree->key)
      Insert(k, btree->left);
    else if (k > btree->key)
      Insert(k, btree->right);
    RightRotate(btree);
    LeftRotate(btree);
  }
}
//-----------------------------------------------------------
/* Inserisce un elemento come radice (chiamata da Insert(T)
   se l'albero e` vuoto)
   INPUT: chiave dell'elemento da inserire */
template <class T> void BBTree<T>:: InsertRoot(T k)
{
  BT = new nodo<T>(k);
  BT->key = k;
  BT->left = bottom;
  BT->right = bottom;
  BT->level = 1;
}

//-----------------------------------------------------------
```

```
/* Cancella un elemento (viene chiamata da Delete(T))
   INPUT: chiave dell'elemento,sottoalbero da cui eliminare */
template <class T> void BBTree<T>:: Delete(T k, nodo<T>*& btree)
{
  if(btree!=bottom)
  {
    // Fase 1: discendi nell'albero impostando "last" (punta
    //         all'ultimo nodo interno incontrato) e "deleted"
    //         (punta all'ultimo nodo da cui si e` scesi a dx,
    //         quello con la piu` piccola chiave maggiore di k)
    last = btree;
    if (k < btree->key)
      Delete(k, btree->left);
    else
    {
      deleted = btree;
      Delete(k, btree->right);
    }
    // Fase 2: giunto in fondo all'albero togli l'elemento,se c'e`.
    //         "deleted" punta al nodo con l'elemento da cancellare
    //         "last" punta al nodo da eliminare
    if (btree==last && deleted != bottom && k == deleted->key)
    {
      deleted->key = btree->key;
      deleted = bottom;
      btree = btree->right;
      delete(last);
    }
    // Fase 3: ribilancia l'albero durante la risalita richiamando
    //         RightRotate e LeftRotate per ogni nodo incontrato
    else if (btree->left->level < btree->level -1 ||
             btree->right->level < btree->level -1)
    {
      btree->level = btree->level -1;
      if (btree->right->level > btree->level)
        btree->right->level = btree->level;
      RightRotate(btree);
      RightRotate(btree->right);
      RightRotate(btree->right->right);
      LeftRotate(btree);
      LeftRotate(btree->right);
    }
  }
}
//--------------------------------------------------------
/* Cerca un elemento nel B-albero binario (viene chiamata da
   Find(T))
   INPUT: chiave dell'elemento,sottoalbero in cui cercare
   OUTPUT: 1 se trova l'elemento, 0 altrimenti */
template <class T> int BBTree<T>:: Find(T k, nodo<T>* p)
```

```
{
  int ret = 0;
  if (p!=bottom)
  {
    if (p->left != bottom)
      if (Find (k, p->left)) ret=1;
    if (k == p->key)
      ret=1;
    if (p->right != bottom)
      if (Find(k, p->right)) ret=1;
  }
  return ret;
}
//-----------------------------------------------------------
/* Trova l'elemento con chiave min del sottoalbero radicato
   nel nodo che le viene passato (la chiama Minimum())
   INPUT: nodo
   OUTPUT: chiave dell'elemento minimo */
template <class T> T BBTree<T>:: Minimum(nodo<T>* p)
{
  while (p->left!=bottom)
    p = p->left;
  return p->key;
}
//-----------------------------------------------------------
/* Trova l'elemento con chiave max del sottoalbero radicato
   nel nodo che le viene passato (la chiama Maximum())
   INPUT: nodo
   OUTPUT: chiave dell'elemento massimo */
template <class T> T BBTree<T>:: Maximum(nodo<T>* p)
{
  while (p->right!=bottom)
    p = p->right;
  return p->key;
}
//-----------------------------------------------------------
/* Visualizza gli elementi del sottoalbero radicato nel nodo
   che le viene passato (la chiama Show())
   INPUT: nodo */
template <class T> void BBTree<T>:: Show(nodo<T>* btree)
{
  if (btree != bottom)
  {
    if (btree->left != bottom)
      Show(btree->left);
    cout << "-- " << btree->key << endl;
    if (btree->right != bottom)
      Show(btree->right);
  }
}
```

```
//--------------------------------------------------------
/* Esegue una rotazione destra
   INPUT: nodo */
template <class T> void BBTree<T>:: RightRotate(nodo<T>*& btree)
// Inclina: Toglie eventuali connessioni orizzontali sinistre
//          effettuando una rotazione destra
{
  nodo<T>* temp;

  if (btree->left->level == btree->level)
  {
    temp = btree;
    btree = btree->left;
    temp->left = btree->right;
    btree->right = temp;
  }
}
//--------------------------------------------------------
/* Esegue una rotazione sinistra
   INPUT: nodo */
template <class T> void BBTree<T>:: LeftRotate(nodo<T>*& btree)
// Fraziona: Modifica un sottoalbero se un nodo e il suo figlio
//           destro hanno una connessione orizzontale destra
//           effettuando una rotazione sinistra

{
  nodo<T> * temp;

  if (btree->right->right->level == btree->level)
  {
    temp = btree;
    btree = btree->right;
    temp->right = btree->left;
    btree->left = temp;
    btree->level = btree->level + 1;
  }
}
//--------------------------------------------------------
#endif
```

Come al solito ci sono due file di esempio d'uso dei B-alberi binari.

14.3 TestTree.cpp

85 ⟨*TestTree.cpp* 85⟩≡

```
/******************************************************************
 *                        B-ALBERI BINARI                         *
 *File di test e di esempio d'uso del B-albero binario realizzato da*
 *          ----------- Maria Luisa Guerriero -----------          *
 ******************************************************************/
/* -----------------------------------------------------
* In questo esempio il B-albero binario e‘ utilizzato per
  contenere delle stringhe (string), ma si potrebbe usare anche per
  inserire altri tipi di dato (ad esempio int, float,...).
----------------------------------------------------- */

#include <iostream.h>
#include <string>

#include "BBTree.h"

//-----------------------------------------------------

void main()
{
 cout<<"Creo un B-albero binario per contenere stringhe(string)\n";
 BBTree<string> *T = new BBTree<string>;
 string key;

 cout<<"Inserisco la stringa: \"prova\"\n";
 key="prova"; T->Insert(key);
 cout<<"Inserisco la stringa: \"uso\"\n";
 key="uso"; T->Insert(key);
 cout<<"Inserisco la stringa: \"B-alberi\"\n";
 key="B-alberi"; T->Insert(key);
 cout<<"Inserisco la stringa: \"binari\"\n";
 key="binari"; T->Insert(key);

 cout<<"Mostro alcune informazioni sull'albero...\n";
 if (!T->IsEmpty())
 {
   cout << "height = " << T->Height() << endl;
   cout << "size = " << T->Size() << endl;
   cout << "radice = " << T->Root() << endl;

   cout<<"...ed il suo contenuto.\n";
   T->Show();
 }
 cout << "Cerco la chiave \"B-albero\"...\n";
```

```
 key="B-albero";
 if (!T->IsEmpty())
 {
   if (T->Find(key))
     cout << "trovata la chiave \"" << key << "\"\n";
   else
   {
     cout << "la chiave \"" << key;
     cout << "\" non appartiene all'albero\n";
   }
 }
 cout<<"Mostro di nuovo il contenuto dell'albero\n";
 if (!T->IsEmpty())
   T->Show();
 else cout << "albero vuoto\n";

 cout << "Cerco la chiave \"B-alberi\"...\n";
 key="B-alberi";
 if (!T->IsEmpty())
 {
   if (T->Find(key))
     cout << "trovata la chiave \"" << key << "\"\n";
   else
   {
     cout << "la chiave \"" << key;
     cout << "\" non appartiene all'albero\n";
   }
   cout << "...e la cancello.\n";
   T->Delete(key);
 }
 cout<<"Mostro di nuovo il contenuto dell'albero\n";
 T->Show();

 cout << "Cancello le altre 3 chiavi...\n";
 T->Delete("prova");
 T->Delete("uso");
 T->Delete("binari");

 cout << "...e controllo che l'albero sia vuoto.\n";
 if (!T->IsEmpty())
   T->Show();
 else cout << "albero vuoto\n";
}
//-----------------------------------------------------------
```

14.4 TestInterattivoTree.cpp

87 ⟨*TestInterattivoTree.cpp* 87⟩≡

```
/***************************************************************
 *                         B-ALBERI BINARI                     *
 *File per un utilizzo guidato del B-albero binario realizzato da*
 *         ----------- Maria Luisa Guerriero -----------        *
 ***************************************************************/
/* -----------------------------------------------------
* In questo esempio il B-albero binario e‘ utilizzato per
  contenere degli interi (int), ma si potrebbe usare anche per
  inserire altri tipi di dato (ad esempio int, float,...).
----------------------------------------------------- */
#include <iostream.h>
#include <string>

#include "BBTree.h"

#define STRLEN 80

//-----------------------------------------------------

int main(void)
{
  char sc;
  int key;
  BBTree<int> *T = new BBTree<int>;

  cout <<"In questo esempio nell’albero sono memorizzati interi";
  cout <<"(int), ma si possono inserire altri tipi di dati";
  cout <<"(ad es. string, float,...)\n";

  while(1)
  {
    cout << "1) Inserire un elemento\n";
    cout << "2) Altezza & Size\n";
    cout << "3) Radice\n";
    cout << "4) Visualizzare l’albero\n";
    cout << "5) Cancellare un elemento\n";
    cout << "6) Cercare un elemento\n";
    cout << "7) Minimo\n";
    cout << "8) Massimo\n";
    cout << "9) Uscire\n";

    cin >> sc;
    switch (sc)
    {
      /* 1) Inserire un elemento */
      case ’1’: cin >>key;
                T->Insert(key);
```

```
          break;

/* 7) Altezza & Size */
case '2': if (!T->IsEmpty())
          {
            cout << "height = " <<T->Height() << endl;
            cout << "size = " << T->Size() << endl;
          }
          else cout << "albero vuoto" << endl;
          break;
/* 3) Radice */
case '3': if (!T->IsEmpty())
            cout << "radice = " << T->Root() << endl;
          else cout << "albero vuoto" << endl;
          break;

/* 4) Visualizzare l'albero */
case '4': if (!T->IsEmpty())
            T->Show();
          else cout << "albero vuoto" << endl;
          break;

/* 5) Cancellare un elemento */
case '5': if (!T->IsEmpty())
          {
            cin >>key;
            T->Delete(key);
          }
          else cout << "albero vuoto" << endl;
          break;

/* 6) Cercare un elemento */
case '6': if (!T->IsEmpty())
          {
            cin >>key;
            if (T->Find(key))
              cout << "trovata la chiave \"" << key << "\"\n";
            else
            {
              cout << "la chiave \"" << key;
              cout << "\" non appartiene all'albero\n";
            }
          }
          else cout << "albero vuoto" << endl;
          break;

/* 6) Minimo */
case '7': if (!T->IsEmpty())
            cout << "minimo = " <<T->Minimum()<< endl;
          else cout << "albero vuoto" << endl;
```

```
                break;

      /* 6) Massimo */
      case '8': if (!T->IsEmpty())
                  cout << "massimo = " <<T->Maximum()<< endl;
                else cout << "albero vuoto" << endl;
                break;

      /* 7) Uscire */
      case '9': exit(0);
    }
  }
}
//-----------------------------------------------------------
```

Appendice A

TABELLE HASH: Tecniche comuni di hashing

Vi sono diversi modi per definire una funzione hash. Ecco i vari approcci che possono essere utilizzati (nell'illustrarli, assumerò che `unsigned char` abbia dimensione di 8-bit, `unsigned short int` di 16-bit e `unsigned long int` di 32-bit):

1. **METODO DELLA DIVISIONE**

   [`HashTableSize` = numero primo]
   Questa è la tecnica usata nell'esempio precedente. Un generico indice `HashValue`, da 0 a (`HashTableSize`-1), viene calcolato dividendo il valore dell'elemento per la dimensione della tabella hash e prendendo il resto. Per esempio:

   ```
   typedef int HashIndexType;
   HashIndexType Hash(int Key) {
   return Key % HashTableSize;
   }
   ```

   Scegliere un valore appropriato per `HashTableSize` è importante per l'efficacia di questo metodo. Per esempio, un `HashTableSize` divisibile per 2 porterebbe a valori di hash pari per chiavi pari, e a valori di hash dispari per chiavi dispari. Questa è una proprietà indesiderata, in quanto, nel caso in cui tutti gli elementi fossero pari, verrebbero portati su valori di hash pari. Se `HashTableSize` è una potenza di due, allora la funzione di hash non fa altro che selezionare un sottoinsieme dei bit di `Key` come indice nella tabella. Per ottenere una distribuzione più casuale, `HashTableSize` dovrebbe essere un numero primo, non troppo prossimo ad una potenza di due.

2. **METODO DELLA MOLTIPLICAZIONE**

   [`HashTableSize` = 2^m]
   Il metodo della moltiplicazione può essere usato per un `HashTableSize` che sia una potenza di 2. Il valore `Key` viene moltiplicato per una costante,

e quindi ne vengono estratti i bit necessari per indicizzare la tabella. In [8] Knuth raccomanda di usare la parte frazionaria del prodotto fra il valore da inserire e il rapporto aureo, cioè $(\sqrt{5}-1)/2$. Per esempio, assumendo che la dimensione di una word sia di 8 bit, si fa:

$$\left\lfloor \frac{\sqrt{5}-1}{2} \cdot 2^8 \right\rfloor = 158$$

Il prodotto fra il valore di 8-bit e 158 corrisponde ad un intero su 16-bit. Se la dimensione della tabella è 2^5 si estraggono come valore di hash i 5 bit più significativi della word meno significativa. Le seguenti definizioni possono essere utilizzate per il metodo della moltiplicazione:

```
/* indice a 8-bit */
typedef unsigned char HashIndexType;
static const HashIndexType K = 158;
/* indice a 16-bit */
typedef unsigned short int HashIndexType;
static const HashIndexType K = 40503;
/* indice a 32-bit */
typedef unsigned long int HashIndexType;
static const HashIndexType K = 2654435769;
/* w=bitwidth(HashIndexType), HashTableSize=2**m */
static const int S = w - m;
HashIndexType HashValue = (HashIndexType)(K * Key) >> S;
```

Per esempio, se `HashTableSize` vale 1024 (2^{10}), un indice a 16-bit è sufficiente, e ad `S` viene assegnato il valore 16 - 10 = 6. In questo modo avremo:

```
typedef unsigned short int HashIndexType;
HashIndexType Hash(int Key) {
static const HashIndexType K = 40503;
static const int S = 6;
return (HashIndexType)(K * Key) >> S;
}
```

3. **METODO DELL'ADDIZIONE PER STRINGHE VARIABILI**

[`HashTableSize` = 256]
Per generare il codice hash di una stringa di lunghezza variabile se ne somma ogni carattere, modulo 256, ad un totale. Il valore `HashValue` risultante varia fra 0 e 255.

```
typedef unsigned char HashIndexType;
HashIndexType Hash(char *str) {
HashIndexType h = 0;
while (*str) h += *str++;
return h;
}
```

4. **METODO DELL'OR ESCLUSIVO PER STRINGHE VARIABILI**

   [`HashTableSize` = 256]
   Questo metodo è simile al metodo dell'addizione, in più è in grado di distinguere parole simili e anagrammi. Per ottenere un valore hash nell'intervallo 0-255, tutti i byte della stringa sono messi in or esclusivo. Inoltre viene introdotta una componente casuale ad ogni or esclusivo effettuato.

```
typedef unsigned char HashIndexType;
unsigned char Rand8[256];
HashIndexType Hash(char *str) {
HashIndexType h = 0;
while (*str) h = Rand8[h \^{} *str++];
return h;
}
```

   `Rand8` è una tabella di 256 numeri casuali unici a 8-bit (il cui ordine non è critico) che può essere facilmente ottenuta con un generatore di numeri pseudo-casuale. Il metodo dell'or esclusivo ha origine dalla crittografia, ed è abbastanza efficiente.

5. **METODO DELL'OR ESCLUSIVO PER STRINGHE VARIABILI**

   [`HashTableSize` $\leq$ 65536]
   Se generiamo due volte il codice hash della stringa, otterremo un valore adatto ad indicizzare una tabella di dimensione arbitraria, fino a 65536. La seconda volta che generiamo il codice hash della stringa, sommiamo uno al primo carattere. I due codici hash a 8 bit vengono quindi concatenati per ottenere un unico codice hash a 16 bit.

```
typedef unsigned short int HashIndexType;
unsigned char Rand8[256];
HashIndexType Hash(char *str) {
HashIndexType h;
unsigned char h1, h2;

if (*str == 0) return 0;
h1 = *str; h2 = *str + 1;
str++;
while (*str) {
h1 = Rand8[h1 ^ *str];
h2 = Rand8[h2 ^ *str];
str++;
}

/* h appartiene all'intervallo 0..65535 */
h = ((HashIndexType)h1 << 8)|(HashIndexType)h2;
/* il valore hash deve essere tra 0 e HashTableSize*/
return h % HashTableSize;
}
```

Appendice B

TABELLE HASH: Metodi a scansione interna

La **scansione lineare a passo unitario** funziona in questo modo: se una chiave z viene associata ad una locazione $hash(z) = h$ che già contiene un altro valore, viene selezionata la locazione disponibile più vicina tra quelle successive all'elemento inizialmente selezionato. Prova cioè a mettere il nuovo elemento alla posizione $h + 1$. Se anche questa è occupata, prova in quella successiva, e così via (tornando alla prima quando si raggiunge $m - 1$). In Figura B.1, se il valore 5960010 viene associato alla locazione 3 dalla funzione hash, linear probe esegue una ricerca sequenziale che scandisce la tabella fino a quando non trova una locazione vuota.

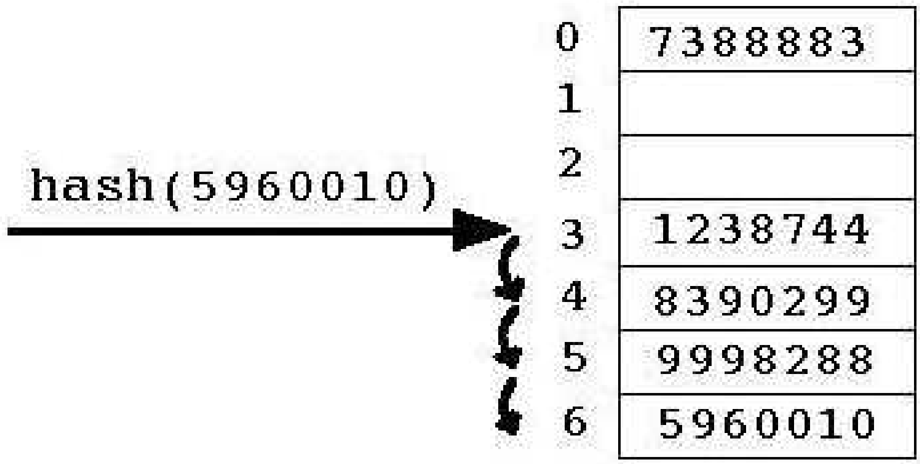

Figura B.1: Tabella Hash con scansione lineare

Questo metodo è semplice, ma in generale non molto efficiente. Immaginiamo di avere molti piccoli gruppi di celle occupate nella hash table, e supponiamo che un nuovo record capiti in uno dei gruppi. Usando la scansione lineare finiremmo per metterlo subito dopo la fine del gruppo, restringendo leggermente il gap tra un gruppo e il successivo. Ad un certo punto due gruppi si fonderanno in uno solo, aumentando così la probabilità che un nuovo record finisca nel nuovo "super-gruppo" e lo faccia crescere ancora. Questo fenomeno degli agglomerati secondari (opposti agli agglomerati primari introdotti da una mediocre funzione hash primaria), aumenta notevolmente il tempo di ricerca.

Il metodo di **scansione quadratica** prova a disperdere gli indirizzi secondari lungo tutta la tabella. Se il primo indirizzo h è occupato, prova in sequenza $h+1, h-1, h+4, h-4, h+9, h-9, h+16, h-16$, e così via (tutti in mod M). Attenzione: se la dimensione M della tabella non è scelta accuratamente, la scansione quadratica scandirà solo metà della hash table e abbandonerà, anche se metà della caselle sono libere: per evitare questo problema M dovrebbe essere un numero primo, con $M = 4 * intero + 3$.

Un altro approccio all'indirizzamento secondario è quello di generare un incremento secondario basato sulla chiave allo stesso modo di come è stato prodotto l'indirizzo primario, ma usando una funzione hash diversa (**hashing doppio**). Questo incremento d è usato per cercare $h + d, h + 2d$, ecc. (sempre in mod M) fino a che non si trova una posizione libera. Se M è primo questo metodo (così come ogni metodo con incremento fisso) prova tutte le posizioni della tabella prima di abbandonare. Un approccio semplice è quello di usare il quoziente $q = z \, div \, M$ come incremento d: la sequenza di scansione secondaria sarà quindi $h + q, h + 2q, ...$, ecc. (mod M).

Le cancellazioni possono causare difficoltà nei metodi a scansione interna: infatti quando in una ricerca si incontra una casella libera significa che la ricerca non ha avuto successo; bisogna perciò distinguere tra caselle libere perché mai riempite e caselle libere perché contenevano un elemento che è stato cancellato. Questo problema non è presente invece utilizzando un metodo a scansione esterna. Per questo motivo la scansione interna può essere usata quando si effettuano solo inserimenti e non cancellazioni.

Bibliografia

[1] Bayer e McCreight, *Organization and Maintainance of Large Ordered Indices*, Acta Informatica 1 (1972)

[2] Bertossi, *Algoritmi e Strutture di Dati*, UTET (2000)

[3] Bistock, *Hashing Rehashed*, DDJ (1996)

[4] Cormen, Leiserson e Rivest, *Introduction to Algorithms*, McGraw-Hill (1990)

[5] Enbody e Du, *Dynamic Hash Schemes*, ACM Computing Surveys (1988)

[6] Foster, *Information storage and retrieval using AVL trees*, Proceedings of the 20th ACM National Conference (1965)

[7] Karlton, Fuller, Scroggs e Kachler, *Performance of height balanced trees*, Communications of the ACM (1976)

[8] Knuth, *The Art of Computer Programming*, Vol. 3: Sorting and Searching, Addison-Wesley (1973)

[9] Larson, *Dynamic Hash Table*, Communications of the ACM (1988)

[10] LoRusso, *Analisi di una Hash Table ideale*, Computer Programming 56 (1997)

[11] McClung, *Utilizzare gli Array Associativi*, Computer Programming (1995)

[12] Nievergelt, *Binary search trees and file organization*, Computing Surveys (1973)